Testing Probabilistic Programming Systems

Saikat Dutta
University of Illinois, USA
saikatd2@illinois.edu

Owolabi Legunsen
University of Illinois, USA
legunse2@illinois.edu

Zixin Huang
University of Illinois, USA
zixinh2@illinois.edu

Sasa Misailovic
University of Illinois, USA
misailo@illinois.edu

ABSTRACT

Probabilistic programming systems (PP systems) allow developers to model stochastic phenomena and perform efficient inference on the models. The number and adoption of probabilistic programming systems is growing significantly. However, there is no prior study of bugs in these systems and no methodology for systematically testing PP systems. Yet, testing PP systems is highly non-trivial, especially when they perform approximate inference.

In this paper, we characterize 118 previously reported bugs in three open-source PP systems—Edward, Pyro and Stan—and propose *ProbFuzz*, an extensible system for testing PP systems. ProbFuzz allows a developer to specify templates of probabilistic models, from which it generates concrete probabilistic programs and data for testing. ProbFuzz uses language-specific translators to generate these concrete programs, which use the APIs of each PP system. ProbFuzz finds potential bugs by checking the output from running the generated programs against several oracles, including an accuracy checker. Using ProbFuzz, we found 67 previously unknown bugs in recent versions of these PP systems. Developers already accepted 51 bug fixes that we submitted to the three PP systems, and their underlying systems, PyTorch and TensorFlow.

CCS CONCEPTS

• **Software and its engineering → Software testing**;

KEYWORDS

Probabilistic programming languages, Software Testing

ACM Reference Format:
Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. 2018. Testing Probabilistic Programming Systems. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3236024.3236057

1 INTRODUCTION

Probabilistic programming has recently emerged as a promising approach for helping programmers to easily implement Bayesian inference problems and automate efficient execution of inference tasks. Both research and industry have proposed various probabilistic programming systems, e.g., Church [40], Stan [35], and many others [8, 16, 36, 46, 61, 71, 120]. These systems automate various parts of common inference tasks and support many approximate inference algorithms from machine learning and statistics, including deterministic variational inference and randomized Markov Chain Monte Carlo (MCMC) simulation. Systems like Edward [31, 119] and Pyro [84] embed probabilistic inference within the general deep learning infrastructures, e.g., PyTorch [85] and TensorFlow [117].

A probabilistic programming system (PP system) typically consists of a language, a compiler, and inference procedures. A programmer writes a program in a probabilistic programming language, which extends a standard programming language by adding constructs for (1) random choice, such as sampling from common distributions, (2) conditioning on data, such as observation statements, and (3) probabilistic queries, such as obtaining a posterior distribution or an expected value of a program variable [41]. Next, a PP system compiles the probabilistic program to an efficient inference procedure, by adapting well-known inference algorithms. Finally, the programmers run the compiled program on a set of data points to compute the query result.

Probabilistic programming systems provide many benefits to programmers who are non-experts in probability and statistics, but ensuring the correctness of probabilistic programs is notoriously difficult [44, 90]. The inherent uncertainty and complexity of probabilistic inference (which is #P-hard, even with just discrete variables [17]) make most practical inference algorithms numerically intensive and approximate. Therefore, a testing approach for PP systems must account for both numerical errors and errors due to the approximate nature of inference algorithms.

Current approaches for testing PP systems are typically manual and ad-hoc. Although recent research looked into analysis of PP systems [1, 90], none of the proposed approaches can analyze all stages of modern PP systems. Understanding previously known bugs in PP systems and finding effective approaches to improve the systems' reliability remain open research questions.

1.1 Bugs in Probabilistic Programming Systems

To motivate the design of tools for systematic testing of PP systems, we characterized the kinds of bugs that are common in existing open-source systems. To the best of our knowledge, this is the first systematic study of bugs in PP systems. We studied three systems: Edward [31, 118, 119], Pyro [84], and Stan [10, 35, 51, 95]. They are written in multiple programming languages, are hosted on GitHub, have been adopted by both industry and researchers, are actively developed, and implement many language features and inference algorithms that are common to most PP systems. In total, we categorized 118 of 856 commits about bugs as being PP systems-related, and describe them in more detail.

Many of the identified bugs required *domain-specific* knowledge to detect, debug, and fix. Moreover, testing PP systems often requires reasoning about result *accuracy* (in contrast to the standard

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236057

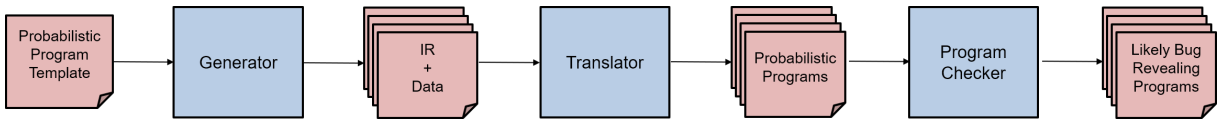


Figure 1: ProbFuzz Architecture

notion of binary pass/fail result correctness). We identified two domain-specific classes of bugs in PP systems: *algorithmic/accuracy bugs* and *dimension bugs*. Algorithmic bugs influence computations of probability expressions and the steps of the inference algorithms, often resulting in decreased result accuracy and are typically hard to identify and fix. Dimension bugs occur when computations do not properly handle the dimensions of data or allowable ranges of probability distribution parameters. Section 3 describes the lessons learned from studying these historical bugs, which we leveraged to design ProbFuzz.

1.2 ProbFuzz

We present ProbFuzz, a novel approach and system for systematic testing of PP systems. Figure 1 shows the architecture of ProbFuzz. The inputs to ProbFuzz are (1) a specification of the primitive discrete and continuous distributions, (2) the number of programs to generate, and (3) a template that specifies the skeleton of a probabilistic program (model) of interest, written in a high-level probabilistic language notation (IR). ProbFuzz outputs a set of programs that are likely bug-revealing in the PP systems. ProbFuzz has three main components:

- *Generator* completes holes in the template to produce (1) a probabilistic program in an intermediate language and (2) accompanying data necessary to run probabilistic inference. Template completion is a form of fuzzing: Generator produces many programs, with different concrete distributions, distribution parameter values, and data values. To generate programs that are more likely to identify non-trivial bugs, Generator incorporates domain-specific information, e.g., legal connections among distributions, ranges of their parameters, and data properties.
- *Translator* converts the intermediate probabilistic program to a specific API or language of a PP system under test, and selects system-supported inference algorithms. We implemented three versions of Translator, for Edward, Pyro, and Stan.
- *Program Checker* runs the generated programs and determines whether the outputs indicate likely bugs in the PP system on which the programs were run. Program Checker produces a set of likely bug-revealing programs for developers to inspect, and supports checks for standard problems (like crashes or `NaN` errors in the output) and accuracy of inference results.

We designed Generator to be general – it represents probabilistic models in the intermediate first-order probabilistic language, and can target various PP systems. We designed Translator to be flexible and extensible. Our experience is that adding support for a new PP system is relatively easy. Moreover, support for multiple PP system in the Translator enables differential testing as an oracle in the Program Checker. ProbFuzz is available at https://probfuzz.com.

ProbFuzz leverages the observation that testing PP systems is similar to the well-studied field of compiler testing. A prominent approach in compiler testing is *compiler fuzzing* [3, 13, 14, 52–54, 56, 113, 123, 124], which randomly generates many test programs and checks whether a compiler produces code (or crashes) and whether generated programs are correct, i.e., produce same results as reference programs. Our study of existing bugs and evaluation of ProbFuzz show the importance of (1) domain-specific knowledge about probability distributions and inference algorithms, (2) joint generation of programs and corresponding data to run inference, and (3) reasoning about accuracy. These traits are out of reach for state-of-the-art compiler fuzzing techniques.

1.3 Results

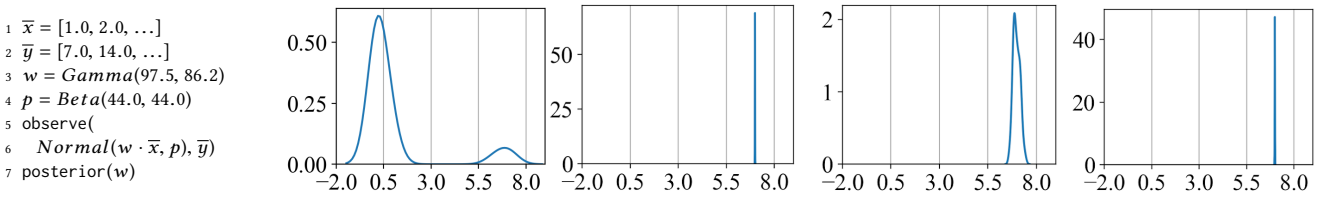
We evaluated ProbFuzz on three PP systems: Edward, Pyro, and Stan. Our evaluation shows the effectiveness of ProbFuzz in generating probabilistic programs and data that reveal dimension/boundary-value and algorithmic/accuracy bugs in all three systems. We discovered 67 potential previously unknown bugs in these systems. Further, we used ProbFuzz to target existing bugs in each PP system we characterized in Section 3, to see in how many categories per PP system ProbFuzz would have caught a bug. ProbFuzz caught at least one existing bug in 8 of 9 categories that we targeted. Section 5 presents quantitative results of ProbFuzz.

As part of our bug discovery and understanding process, we submitted all 67 potential bugs revealed by ProbFuzz to developers of the PP systems. So far, developers have accepted 51, rejected 8, 7 are still pending and 1 was already fixed before we could submit it. The bugs that we found and fixed were not just in Edward, Pyro and Stan, but also in the underlying software infrastructure on which they are built (i.e., PyTorch for Pyro, and TensorFlow for Edward). We describe some of the identified bugs, their fixes, and lessons that we learned in Section 6.

1.4 Contributions

This paper makes the following contributions:

- ★ **Concept.** We extend compiler fuzzing to probabilistic programming systems. We generate both probabilistic programs and data to run inference by encoding domain knowledge and reasoning about accuracy of inference results.
- ★ **Bug Characterization.** We present the first study of bugs in PP systems. Our investigation of 118 previously fixed bugs in three open-source PP systems showed that these bugs require domain knowledge to find and fix, and to reason about accuracy.
- ★ **Methodology and System.** We propose ProbFuzz, a novel general approach for systematically testing PP systems. Our current implementation of ProbFuzz works for three open-source PP systems and is extensible.
- ★ **Evaluation.** We evaluated ProbFuzz on both historical and recent versions of Edward, Pyro and Stan. ProbFuzz found bugs in each category of previously reported bugs. We also found and reported 67 previously unknown bugs by running ProbFuzz on recent versions of the PP systems.



(a) Probabilistic Program **(b) Stan Result** **(c) Edward Result** **(d) Pyro Result** **(e) BUGS Result**

Figure 2: Example Program and the Posterior Distributions Computed by Various Systems

2 ILLUSTRATIVE EXAMPLE

Figure 2a shows an illustrative example of a potentially bug-revealing probabilistic program generated by ProbFuzz. The program, shown in ProbFuzz's intermediate language, defines two data-sets of constants, $\overline{x}$ and $\overline{y}$. Each $\overline{y}_i$ is seven times the value of $\overline{x}_i$. The program first assigns prior distributions to the variables w and p. Then, it conditions the linear model $w \cdot \overline{x} + p$ on the observations of $\overline{y}$. The probabilistic query on line 7 seeks the posterior distribution w.

Probabilistic inference is a procedure for computing the change in the distribution of variables based on the observations of data. Most inference algorithms today are approximate, with the two dominant approaches being *Markov Chain Monte Carlo* simulation, which re-executes the computation with many random samples (and is implemented in, e.g., Stan and Edward) and *variational inference*, which approximates the posterior distribution deterministically, by substituting it with computationally simpler distributions (and is implemented in, e.g., Edward and Pyro).

Figures 2b-2e show the posterior distributions computed by Stan, Edward, Pyro, and another probabilistic inference system called BUGS (which is a precursor of Stan, and shares most of its syntax). The X-axis presents the numerical values and the Y-axis presents its probability density function. Given the data $\overline{x}$ and $\overline{y}$, we expect the mean of the posterior of w to be equal to 7.0. The posterior distributions computed by three systems are similar, and centered at 7.0. However, Stan's distribution has a different shape, and its mean is close to 1.0. We discuss the reasons behind this accuracy problem in Section 6.2.

ProbFuzz generates the program in Figure 2a, and many similar programs, with different prior distributions, distribution parameters, and data. ProbFuzz then compiles the programs down to each PP system, generating specialized API calls or DSL programs. The translation is non-trivial, and cumbersome for a human, but can be easily specified in ProbFuzz. Next, ProbFuzz runs generated programs, automatically compares the output from different PP systems, and computes accuracy metrics (Section 4.4). Finally, a developer can inspect ProbFuzz results and investigate any potential bugs. We discuss ProbFuzz in Section 4.

3 BUG CHARACTERIZATION STUDY

We characterized existing bugs in three open-source PP systems: Stan [10, 35, 51, 95], Edward [31, 118, 119] and Pyro [84]. Table 1 shows some statistics about the PP systems. The three PP systems support various approximate probabilistic inference algorithms.

Methodology. We manually searched for bug fixes among commits in the GitHub repositories of the PP systems in our study. We use commits to get a larger data set than we could get when starting from GitHub issues [2, 86]. Given the active development of these PP systems, many bugs are fixed without first being reported as "issues", and most closed issues involve one or more commits.

We obtained all commits in the three PP systems that contained the keywords, `bug|inference|error|fix|nan|exception|overflow|underflow|infinity|infinite|precision|unstable|instability|ringing|unbounded|roundoff|truncation|rounding|diverge|cancellation|cancel|accuracy|accurate`. This resulted in 1837 commits. We then filtered out commits that are not specific to the domain of PP systems or probabilistic inference, and could occur in any software domain. First, we filtered out commits containing the following keywords: `typo|docstring|notes|example|examples|tutorial|print|doc|Document|messaging|test|messages|manual|doxygen|cpplint|Jenkins|submodule|header`. Next, we split the remaining 856 commits between two student coauthors, each of whom read descriptions and reasoned about modified code. Each coauthor marked a commit as an inference-related code fix, general code fix, a refactoring, or a duplicate. We filtered out refactoring, duplicates (e.g., covered by incremental commits fixing the same bug or related commits from multiple branches), merge commits with many files changed, and commits that changed only non-source files.

We were left with 455 commits that fix code, out of which our manual inspection identified 118 commits that are directly related to the domain of probabilistic inference. The remaining are general coding problems e.g., I/O errors, API misuses, and documentation problems. Two coauthors inspected these 455 commits. They compared notes and classified bugs as inference-related only if they agreed on the final classification, therefore making a conservative determination about the domain-specific nature of each bug. Similar to a previous work on analyzing numerical bugs [19], we put inference-related bugs into four categories. Our bug categories are algorithmic/accuracy, dimension/boundary-values, numerical, and language/translation. We made a second pass through the 118 bugs that satisfy the selection criteria and categorized them based on error sources and bug manifestations. When possible, we matched each commit to its related GitHub issue.

Table 1: Project Statistics

	Edward	Pyro	Stan
First commit date	Feb 10 '16	Jun 15 '17	Sep 30 '11
No. of contributors	74	26	61
No. of commits	1780	853	13083
Latest commit studied	992ce08	8db8972	14981a3
Lines of code	12035	11609	57770
Prog. language	Python	Python	C++
Infrastructure	Tensorflow	PyTorch	Own

Table 2: Breakdown of Commits

Category	Edward	Pyro	Stan	∑
Algorithmic/accuracy	9	10	16	35
Dimension/boundary	11	14	13	38
Numerical	1	1	17	19
Language/translation	5	7	14	26
∑	26	32	60	118

3.1 Characterizing Bugs in PP Systems

Table 2 shows the distribution of the categorized commits. Column "Category" shows category names. The second to fourth columns show the number of commits per category in each PP system. Finally, column ∑ presents the sum of the commits in each bug category. The database of inference-related bugs is available at https://probfuzz.com/db.

Algorithmic/accuracy bugs. This category contains bugs due to incorrect implementation of inference algorithms and other related bugs in the implementations of probability distributions and statistical procedures. They manifest as inaccurate, although plausible (and therefore hard to catch) results of inference. These bugs affected a variety of inference algorithms and implementations of probability distributions in all three PP systems. In Edward, the bugs affected three inference algorithms and two built-in distributions (Bernoulli and Uniform). In Pyro, the bugs affected three inference algorithms and the Cauchy distribution. In Stan, the bugs affected two inference algorithms, one distribution (Bernoulli Logit) and two auxiliary functions.

These bugs can be further subdivided into logical errors, mathematical errors, and one regression error. Examples of logical bugs include re-normalizing already normalized data [28], "double-counting" the values of specific variables [81], and using only the first element instead of a whole collection to fill a tensor [75]. Examples of mathematical errors include incomplete formulae (e.g., missing terms [26, 77]) and wrong formulae (e.g., [30]). Finally, a regression in Stan led to lower statistical efficiency [104].

Dimension/boundary-value bugs. These bugs occur when functions do not properly handle the dimensions of input data (a scalar, vector, matrix, or cube), the ranges of input data, and the ranges of distribution parameters. They manifest as exceptions or special numerical values, e.g., `NaN` or `Inf`, in the output (in the case of boundary-value bugs). The examples of dimension bugs include those where the functions assumed a particular dimension of input data (e.g., scalars [25]) and crash if data with different dimension is passed as input, or assumed a wrong dimension of output which caused crashes in the function's clients (e.g., [83]). One bug resulted from using only one ordering of a list (a vector) to compute entropy, instead of using all possible orderings (a matrix) [21].

Missing boundary condition checks often happen in implementations of various probability distributions, e.g., not checking for boundary values of a parameter leading to `NaN` [82]. Such bugs typically manifest substantially late during inference, e.g., computing `log` of zero resulting in `NaN` [80]. We also observed some off-by-one errors (e.g. in [100, 107]), where `if` conditions used $<$ instead of $\leq$.

General numerical bugs. These bugs are found in general mathematical functions, and may manifest as an inaccurate result or a special value (`NaN` or `Inf`). Most of these bugs are in Stan, which implements its own mathematical back-end, in contrast to Edward and Pyro, which use external back-ends (TensorFlow and PyTorch, respectively). Example numerical bugs that we identified include improper handling of `Inf` (e.g., [23, 78]) or `NaN` (including when these special values propagate to the output [105]), initializing `Integer` values to `NaN`, overflow errors, and convergence bugs.

Language/translation bugs. These bugs occur due to wrong use of features in the programming language in which the PP system is written. They can manifest as failed builds, runtime errors, or wrong results. These can be errors in the interface (e.g., [99], returning a `real` instead of an array as expected from the API specification), errors in the back-end or changes in their implementations (e.g., [30]), errors that break compilation or error reporting (e.g., [106]), and errors in using functionality. One functionality usage error involved calling a stateful inference function, making different runs of the same probabilistic program producing widely different results [22].

3.2 Discussion

We highlight several important observations from our characterization study, which motivate our approach for testing PP systems:

Observation 1: Domain knowledge is required to detect, analyze, and fix bugs. Most of the inspected Algorithmic/accuracy and Dimension/boundary-value bugs, and some Numerical bugs require knowledge of theory of probability or inference. Bugs in the Dimension/boundary-value category are similar to general bugs that occur when one does not satisfy the specification of a method. However, without specification-related assertions (which require domain-specific knowledge, and are tedious to write) in the code, such bugs occur in the PP systems, resulting in `NaN` or silent errors.

Observation 2: Algorithmic bugs require detailed reasoning about accuracy. For many of the inference and accuracy bugs, the developers report (in)accuracy of the results and compare the results either to known (expected) values or against another tool (e.g., Edward or Pyro against Stan). For algorithmic errors, existing numerical analyses [57, 87] are typically not applicable. Identifying errors and their causes requires probabilistic reasoning, detailed error reports and discussions with PP system developers in order to diagnose the error (e.g., [108]).

Observation 3: Testing PP systems requires careful generation of both programs and valid data. Reproducing many of the bugs that we manually inspected required both a probabilistic program and the data to run it on. The GitHub issues related to the commits that we inspected had *both* programs (or program fragments) and data necessary to reproduce the bug. Such data is sampled from probability distributions and is required for setting up priors and posteriors, distribution parameters, and as inputs for inference. This is different from compiler testing [3, 13, 14, 53, 113, 123, 124], where it is sufficient to simply generate programs that take no inputs and encode arbitrary scalar values of variables.

Observation 4: Many errors are revealed by small programs. Most GitHub issues related to the commits that we inspected had small reproducible programs. The observation that many bugs can be found by small programs is well-known [48], and has been used extensively in conventional testing. While standard compiler testing (e.g., CSmith [123]) often generates large programs to maximize bug-finding capability, small programs seem sufficient for successful detection and debugging in the PP system domain.

4 PROBFUZZ

ProbFuzz takes as inputs the template of the probabilistic model, the number of programs to generate and the systems to test. The developer writes the templates of probabilistic models in an intermediate probabilistic language with holes, which represent missing distributions, parameters, or data (Section 4.1).

Figure 3 presents the pseudo code of the ProbFuzz algorithm. The Generator generates probabilistic models by completing holes in the template with concrete distributions, parameters and data (Section 4.2), resulting in a program in an intermediate language. The Translator then translates the probabilistic program from the intermediate language into a program that uses the API of the target PP system (Section 4.3). Next, ProbFuzz runs the programs, collects output, and its Program Checker computes metrics and checks for symptoms that may reveal potentially buggy programs (Section 4.4). Finally, ProbFuzz reports any warnings issued by the Program Checker to the developer.

4.1 Template and Intermediate Language

ProbFuzz represents the templates and the generated programs in an intermediate language (IR). Figure 4a presents the syntax of the IR language of ProbFuzz. The key aspect of the template is *a hole*, denoted as "??". It represents a missing distribution or parameter. The distributions and parameters are completed with concrete values (from respective sets *Dists* and *Consts*) by replacing the hole.

A template consists of four sections, which specify data, prior distributions, model that relates posterior and prior distributions, and the query. The data section presents the input and the output data set(s). A data vector is a typed (multidimensional) array, which is instantiated by ProbFuzz, or a specific list of numerical constants. The *Prior* section specifies the prior distributions of the program variables. A prior distribution can be an instance of a distribution or a hole. Similarly, one or more parameters of the distribution can be either expressions or holes. The expressions are typical, with arithmetic and comparison operators. The language is similar to the loop-free fragment of the Prob language from [41].

The *Model* section conditions the random variables to the specific observations. The `observe` clause states that the observations of the model specified as the first parameter are found in the vector denoted as the second parameter (as is a standard interpretation in most probabilistic languages). The models can also be composed using conditionals. Finally, the *Query* instructs the probabilistic language to return the marginal posterior distributions for the specified variables, or their expected values.

Examples. Figure 4b presents a template from our experiments and Figure 4c presents an example program that has the holes completed. The template is for a linear regression model, which has two sets of observations x and y (both are one-dimensional vectors of length 10). The prior parameters are weight w, bias b, and the noise p, with unknown distributions. The template conditions an unspecified distribution with two parameters (the first is the linear expression $w \cdot \overline{x} + b$, the second is p) on the data from the vector $\overline{y}$.

Distribution Specification. For each distribution, ProbFuzz specifies its properties, including the names and ranges of parameters and the range of the distribution support. Knowing the properties of the distributions allows ProbFuzz to complete the template with the concrete values of parameters.

To illustrate, the specification of the Normal distribution is:

```
"name" : "Normal",
"type" : "Continuous",
"support" : "Float"
"args" : [ { "name" : "mu", "type" : "float"},
           { "name" : "sigma", "type" : "float+" } ],
```

It specifies that the distribution is continuous and its support (the range of values that can be sampled from the distribution) is not constrained. It has two parameters, the mean `mu` is an arbitrary floating-point value, while the standard deviation `sigma` must be positive. The support and parameters of the distribution can be bounded. For instance, in the case of Gamma distribution, the support is only positive real numbers, and in the case of Bernoulli, the support is `{0,1}`.

4.2 Generator

The Generator generates a concrete program and data from the provided template. A concrete program consists of complete IR and data. In a concrete program, all "??" symbols have been replaced by the corresponding distribution expressions or constant expressions (as in Figure 4c). The user-defined program templates plus domain knowledge about distributions and data ranges enable Generator to achieve more targeted fuzzing.

The Generator has two components, the *distribution selector*, which matches the distribution expressions with holes ("??") in the template and the *data selector*, which produces the concrete values of the parameters of the distributions and computes the values of the data points. For each generated program, the Generator performs the following steps:

- **Complete the distribution of the model.** For the model expression, the distribution selector finds all the distributions that can match the pattern (e.g., have two parameters) and uniformly at random selects one of those distributions to fill in the hole. Once fixed, this distribution provides the legal values for the data to generate (based on the distribution support) and the constraints on the parameters. This bounds the set of allowed distributions of the priors in the template. For instance, if we select the Normal distribution for the linear regression template (Figure 4b), the model constrains the distribution of the variance p to have positive support.
- **Complete the distributions of the priors.** Based on the constraints from the model, the distribution selector randomly selects a distribution whose support satisfies the range of values admissible by the model's parameter. To propagate the information about distributions, we implement a simple dependence analysis with interval analysis to keep track of the ranges. For instance, in Figure 4 the distribution selector may choose Exponential as the distribution of the prior for p, but not Normal (since its support is all floating-point values, but p can have only positive values).
- **Complete the distribution parameters.** Data selector picks the numerical values of the parameters of the distributions with holes using a method that randomly chooses between two strategies. The first strategy randomly selects a value within the range of the parameter, as denoted in the distribution specification. A developer may express preference for larger or smaller values

INPUTS: Program count n, Template t, PP systems under test S
OUTPUT: Likely bug-revealing programs report P

function ProbFuzz(n, t, S)
  $P \leftarrow \emptyset$
  for i = 1 to n **do**
    $prog_{IR}$, $data \leftarrow Generator(t)$
    $Results \leftarrow \emptyset$
    for $s \in S$ **do**
      $prog_s \leftarrow Translator_s(prog_{IR}, data)$
      $status_s$, $out_s \leftarrow ExecuteProgram(s, prog_s, data)$
      $Results \leftarrow Results \cup \{(status_s, out_s)\}$
    end for
    $warnings \leftarrow ProgramChecker(Results)$
    if $warnings \neq None$ **then**
      $P \leftarrow P \cup \{warnings\}$
    end if
  end for
  return P
end function

Figure 3: ProbFuzz Algorithm

x	$\in$	*Vars*
c	$\in$	*Consts* $\cup \{-\infty, \infty\}$
aop	$\in$	$\{+, -, *, /\}$
bop	$\in$	$\{=, >, \ldots\}$
$Dist$	$\in$	{ Normal, Uniform, Beta, ...}
Type	::=	`Int` | `Float` | `Range<c, c>` | Type[c]
Data	::=	x : Type | x := $[c^+]$ | x := Expr
Expr	::=	c | x | Expr *aop* Expr | Expr *bop* Expr
Param	::=	?? | Expr
Prior	::=	x := ?? | x := Dist(Param$^+$)
Model	::=	`observe`(Dist(Expr$^+$), x) | `observe`(??(Expr$^+$), x) | x = Expr | `if` (Expr) `then` Model `else` Model
Query	::=	`posterior`(x^+) | `expectation`(x^+)
Template	::=	Data$^+$ Prior$^+$ Model$^+$ Query

(a) Grammar for Probabilistic Program Templates

$\overline{x}$: `Float`[10]
$\overline{y} := c_1 * \overline{x} + c_2$
w =??
b =??
p =??
`observe`(??($w \cdot \overline{x} + b$, p), $\overline{y}$)
`posterior`(w, b, p)

(b) Linear Regression Template

$\overline{x} := [1.0, \ldots]$
$\overline{y} := [2.0, \ldots]$
$w = Gamma(0.3, 5.2)$
$b = Normal(0.3, 2.1)$
$p = Exponential(1.2)$
`observe`($Normal(w \cdot \overline{x} + b, p)$, $\overline{y}$)
`posterior`(w, b, p)

(c) Linear Regression Example

Figure 4: Grammar and Example for ProbFuzz Input Templates

to be inserted here. The second strategy randomly picks values that are close to the boundary values of the parameter ranges; these values may be either legal or illegal and can stress-test the sensitivity of PP systems to boundary conditions and numerical instabilities. The developer can provide a probability that prefers one strategy over the other. For instance, in Figure 4c, data selector picks the values 0.3 and 5.2 as the parameters of the Gamma distribution in the prior of w. Similarly, it could try generating programs where the second parameter of Gamma (which should be positive) is 0.0 or -1.0 to test the capability of the PP system to identify wrong values.

- **Generate the inputs/outputs.** Data selector uses the input range and formulas provided by the developer to compute expected outputs. It then randomly generates the desired number of elements in the input vectors and computes the values in the output vectors.

4.3 Translator

Translator produces a legal program in the language of the target PP system. The inputs to the Translator are the concrete program and data produced by Generator. Each PP system has its own Translator. In addition, Translator takes a configuration file with the list of inference algorithms and a mapping of distributions to corresponding PP system-specific API calls.

Translator in Edward. First, the Inference Selector chooses an inference algorithm that the PP system supports, based on the concrete specification. Second, the Translator replaces distribution names in the input programs with the corresponding API call in Edward, and creates one AST node each for the input data ($\overline{x}$), the model in the program, and the selected inference. Third, several AST nodes are created for the following: (1) one node for the posteriors or each prior, depending on the inference algorithm to be run, (2) a node for a placeholder for $\overline{x}$, and (3) (optional) one node for the proposals for each prior, which is needed for some inference algorithms, e.g., the Metropolis-Hastings (MH) sampling algorithm. Fourth, a `dict` node is created which connects the node for each prior to its respective proposal and posterior nodes, and a `dict` node is created which connects nodes for the data placeholder and the output data, $\overline{y}$. Fifth, the `dict` nodes from the last step are merged with the node for inference. Sixth, the data node, the model node and the inference node are combined together as the final AST. Finally, this AST is converted into a Python program.

Translator in Pyro. The first two steps in the Translator are the same as for Edward: select inference algorithm, replace distributions with corresponding API calls and make AST nodes for $\overline{x}$, the model and the selected inference. The third step is to create a function node for a *Pyro model*, a combination of posterior nodes for each prior which are then connected to the data node. Then a function node for a *Pyro guide* is created with a posterior node for each prior. Next, if the selected inference algorithm is a variational algorithm, an optimization algorithm is chosen together with its parameters based on the concrete specification, and a node is created. Finally, a node for running the inference is created. The generated AST is converted to a Python program.

Translator in Stan. Stan's Translator does not create ASTs. Rather, each model is translated line by line to Stan code stored in `model.stan` file, with data stored in `data.json` file. Finally, a file, `driver.py` is generated and used to run the Stan model.

4.4 Program Checker

The task of the Program Checker is to decide whether output from running the generated programs may be indicative of bugs in the PP system on which the program was run. For Edward and Pyro, the generated Python programs are run directly. The `driver.py` script is run for Stan. Program Checker performs a battery of checks, inspired by the bugs from our characterization study (Section 3):

- **Crash checks**: they find problems with unexpected termination or assertion failures. Crash checks will output programs which crash as likely bug revealing, since all programs generated by ProbFuzz are syntactically and semantically valid.
- **`NaN` and overflow checks:** they will report programs that neither crash nor produce exceptions, but contain `NaN` as output values; as observed in Section 3, they are often related to numerical

and boundary checking problems. Programs which produce `NaN` as output values are potentially bug revealing because it means that the PP system allowed invalid computations to "succeed", instead of warning the developers.

- **Performance checks:** they report if one PP system converges much slower than other PP system.
- **Differential testing with exact result:** these checkers aim to identify accuracy bugs by comparing the results of approximate inference with the exact result. The exact result can be obtained in two ways: (1) using optional data generators , or (2) using exact inference engine. For exact inference, we translate programs to PSI [33]. Exact inference (when it scales) removes approximation and numerical errors, modulo bugs in the exact inference tool. This approach works when the generated programs are small.
- **Differential testing with approximate results:** these checkers aim to identify accuracy bugs by comparing the differences in the results produced by (1) different tools and (2) different algorithms within a single tool or even different versions of the same algorithm (e.g. [108]), and (3) different interfaces to the same inference algorithm. Result comparison across tools or algorithms is useful for accuracy and numerical bugs. Comparisons across different interfaces of the same PP system (e.g., RStan, PyStan) can primarily help find language/translation bugs. The Program Checker issues a warning about a program from which the results of one approximate-inference PP system differs significantly from all other approximate-inference PP systems and the other systems produce similar outputs, or if the outputs of all approximate inference differ from the expected output.

Accuracy Comparisons. Analysis of accuracy is a key challenge in testing PP systems. The computations have various sources of noise: some inference algorithms are randomized (e.g., MCMC), while others make algorithmic approximations (e.g., variational inference). In both cases, there may be rounding errors or overflows.

To quantify the magnitude of errors, ProbFuzz allows a developer to specify custom comparison metrics. In this paper, we compute an accuracy metric based on relative error of the mean. *Symmetric Mean Absolute Percentage Error* [115] computes the distance between the means of the posterior distributions computed by two systems (or comparing the result from one system to the exact result). It is computed as:

$$SMAPE(x_1, \ldots, x_n, y_1, \ldots, y_n) = \frac{1}{n} \sum_{i=1}^{n} \frac{|x_i - y_i|}{|x_i| + |y_i|}$$

The arguments $x_1, \ldots, x_n$ are the means produced by the first system and $y_1, \ldots, y_n$ are the means produced by the second system. In contrast to the usual relative error, which divides the difference by the value from one of the systems, SMAPE does not prefer the result of any of the systems, and is always guaranteed to produce a result in the range $[0, 1]$.

A program may have an accuracy bug if the value of the metric is above a threshold (which effectively acts as a knob for how many programs to inspect). If so, ProbFuzz reports the generated program as revealing a potential accuracy bug. When more than 2 systems are involved, we do a pairwise comparison. If only one of the PP systems shows a significant deviation from the others, ProbFuzz reports that system as likely faulty.

5 QUANTITATIVE EVALUATION

We describe the research questions that we answer, our experimental setup and the quantitative aspects of the results in this paper. We answer the following research questions:

RQ1 How many *new* bugs per category does ProbFuzz find?
RQ2 How many categories of *existing* bugs does ProbFuzz find?
RQ3 How sensitive is the accuracy metric to the threshold choice?
RQ4 How does ProbFuzz compare with conventional fuzzing?

Experimental Setup. For our experiments, we used four templates. We discussed *linear regression* template in Section 4.1. Other templates include *simple posterior template*, which samples from a single distribution, with a prior for each of its parameters and conditions on data, *conditional template*, which chooses between two models based on the if expression, and *multiple linear regression* template with a weight vector for the prior instead of scalar as in linear regression and conditioned on 2-dimensional data sets. We also varied the data vector sizes. We generated 1000 programs per template for each tool. We group the programs based on the determination that Program Checker makes, and then randomly sample a subset of programs in each class for manual inspection. To find performance bugs, we randomly sampled for manual inspection the programs that did not run to completion in the default time-out limit of `3 mins`. For accuracy bugs we used the accuracy metric discussed in Section 4.4 to select wrong programs to manually inspect. The threshold for `SMAPE` that we used in selecting the programs for our manual inspection was `0.1`. We ran all experiments on an Intel Xeon 3.60GHZ machine with 6 cores and 32GB RAM.

5.1 RQ1: New Bugs Discovered by ProbFuzz

Table 3 shows the number per category of the new bugs found during our evaluation of ProbFuzz. Columns (except $\sum$) are the PP systems in our study, while the rows (except $\sum$) are the various categories for which found some bugs that we found. Bug categories were described in Section 3. We counted as bugs either as the number of distinct code locations where we made a fix in pull requests, or one bug for each issue that we submitted to the developers without a corresponding pull request. Note that, by counting each (yet-to-be-fixed) submitted issue as one bug, we are under counting the number of bugs in the code, and the actual number of bugs that ProbFuzz found in our experiments is likely higher.

We submitted 15 issues (each one counts as one bug), and 7 pull requests which fixed 51 bugs in the code. The results show that the dimension/boundary-value bugs are the most common among the bugs that we found. We provide more details in Section 6.1 about how prone the PP systems are to dimension/boundary-value bugs. Among the PP systems, we found the least number of bugs in Stan, followed by Edward and then Pyro. Interestingly, this matches the maturity of the PP systems. We also discuss in Section 6.1 one step that Stan developers have taken over the years to reduce the amount of bugs in this category.

One key benefit that ProbFuzz provides in the testing of PP system is the ability to find accuracy bugs, and not just bugs that lead to crashes or invalid values (e.g., `NaN` or `Inf`) in the output. Accuracy bugs are much more tricky to find and debug; coming up

Table 3: New Bugs per Category Discovered by ProbFuzz

Category	Edward	Pyro	Stan	∑
Algorithmic/accuracy	2	1	2	5
Dimension/boundary	13	41	0	54
Numerical	0	0	3	3
Language/translation	1	3	1	5
∑	16	45	6	67

with oracles that catch them is quite involved and requires domain knowledge. As shown in Table 3, we found 5 potential accuracy bugs in all three PP systems during our manual inspection.

We reported all the bugs in Table 3 to the developers of each PP system. So far, developers have accepted 51, rejected 8, 7 are still pending and 1 was already fixed before we could submit it; 30 accepted bugs were in a single pull request to PyTorch.

5.2 RQ2: Old Bugs Rediscovered by ProbFuzz

This experiment checks whether ProbFuzz can catch a variety of previously fixed bugs that we identified during our characterization study (Section 3). For each PP system, we attempted to reproduce at least one bug per category, such that they cover all categories of interest (Algorithmic/accuracy, Dimension/boundary-value, and Numerical). We did not target Language/Translation bugs, which are specific to each PP system and targeting them requires more involved back-ends. We first checked if these bugs may be reproduced by re-running the tests that failed due to the bug or programs in the corresponding GitHub issue. We stopped if we could no longer run the tests/programs. We did not try to reproduce bugs that cannot be exercised by our four templates. Since some older versions of the PP systems use different syntax and API to specify models or have since undergone major changes, we had to create four additional versions of Translator (for bugs [24, 25, 81, 105]). In addition, we found the versions of the infrastructure (PyTorch and TensorFlow) which were in use in the older versions. For accuracy and numerical bugs, we manually reasoned whether the difference was caused by the bug.

Table 4 shows the numbers and links to bugs that we successfully reproduced with ProbFuzz. For each of these bugs, ProbFuzz generated a program and the data to exercise it. Each cell contains the bug count in each category per PP system. In addition, each cell contains the exact reference to the commit with the bug fix.

The results show that ProbFuzz successfully found bugs in eight out of nine categories of interest. Out of these bugs, six ([24, 25, 76, 79, 97, 101]) were found using the simple posterior template, three using the linear regression template [27, 81, 105] and one using multiple linear regression template [102]. Overall, these results demonstrate that ProbFuzz could have caught a variety of existing bugs, had it been available prior to the discovery of those bugs. Comparison of Tables 3 and 4 shows that ProbFuzz was able to reproduce existing bugs in categories where we did not find any new bug on recent versions of the PP systems (e.g, Stan-Dimension/boundary). Also, ProbFuzz reproduced the only previously known numerical bugs in Edward and Pyro from our characterization study.

5.3 RQ3: Sensitivity of Accuracy Threshold

The number of programs to inspect depends on the threshold set for the accuracy metric. Figure 5 presents the sensitivity of the number of programs reported to potentially reveal accuracy problems as a function of the bound on the SMAPE metric for the linear regression template. The X-axis presents the threshold. The Y-axis presents the fraction of the programs whose accuracy metric (compared to the reference solution) is above the threshold. For the computation, we removed (1) the programs that crashed, (2) the programs that resulted in NaN, and (3) the programs that timed out.

Table 4: Old Bugs per Category Rediscovered by ProbFuzz

Category	Edward	Pyro	Stan	∑
Algorithmic/accuracy	1 [27]	1 [81]	0	2
Dimension/boundary	1 [25]	1 [76]	2 [98, 102]	4
Numerical	1 [24]	1 [79]	2 [101, 105]	4
Language/translation	n/a	n/a	n/a	n/a
∑	3	3	4	10

The results show that the threshold value can serve as a knob for the fraction of the programs to return. For instance, if the threshold is 0.8, then the number of programs with large accuracy loss is less than 10% for Pyro and Stan, and around 14% for Edward. Stan shows an interesting trend of having many programs that have small accuracy loss of the mean, while Edward and Pyro have more programs that have larger accuracy differences.

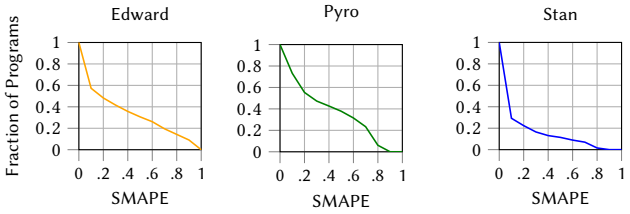


Figure 5: Sensitivity of Accuracy Metric

5.4 RQ4: Benefit of Domain Knowledge

We compared our results with an "uninformed" fuzzer that does not utilize domain knowledge about distributions and legal ranges. Table 5 shows the detailed comparison for 1000 generated programs per tool per template. Each cell contains the percentage of generated programs that produced results without crashing, numerical errors, or timeout for Uninformed ('U') or Informed ('I') fuzzer. On average, less than 21% (Stan 6.35%, Edward 49.07%, Pyro 5.82%) of programs generated by the uninformed fuzzer produce useful results, compared to over 84% (Stan 89.35%, Edward 80.3%, Pyro 83.2%) with ProbFuzz. As such, uninformed approach can be fit for boundary-condition bugs, e.g., when a system fails to recognize a program with wrong values, but it will not be efficient for bugs that can be revealed by only well-formed probabilistic programs.

Table 5: Comparison of Informed vs Uninformed Fuzzing

Template	Stan		Edward		Pyro	
	U	I	U	I	U	I
Simple	16.6	93.4	52.5	80.2	14.0	87.2
LR	6.6	94.3	43.3	80.7	5.2	79.7
MLR	1.2	79.6	43.8	81.2	4.1	83.3
Conditional	1.0	99.0	56.7	79.1	0.0	82.6

6 QUALITATIVE ANALYSIS

During our development and evaluation of ProbFuzz, we encountered several potential bugs in PP systems for which we created fixes and submitted pull requests to the developers. We made fixes in Edward, Pyro, Stan, and also contributed patches to the underlying frameworks, PyTorch and TensorFlow. We present interesting cases, lessons learned, and developer responses.

6.1 Dimension/Boundary Bugs are Common

Dimension/boundary-value bugs accounted for 54 previously unknown bugs, and 38 bugs in our characterization study. In Pyro, we found 41 bugs in this category. One of these bugs in Pyro would lead to a crash whenever the input data is of size `1`; another bug caused an overflow in the Adam Optimizer. We also found similar, previously unknown Dimension/boundary-value bugs in Edward: four bugs were also due to failure to check that parameter values are in the correct range. Interestingly, ProbFuzz did not find any previously unknown dimension/boundary-value bugs in Stan, despite the fact that our characterization study revealed eight dimension/boundary-value bugs that were previously reported in Stan. We attributed this to Stan's relative maturity, compared with Pyro and Edward. Indeed, since March 2014, Stan developers have added a milestone to every major release with the title, *"make sure all distributions throw exceptions at undefined boundaries"* [9].

Lesson Learned: The similarity of dimension/boundary-value bugs found across PP systems suggests that that these bugs are commonly introduced by the developers of the PP systems that we studied. Going forward, developers should continuously test their probabilistic codes for this kind of problems. Automated testing, such as ProbFuzz, can be quite effective for these problems.

6.2 Accuracy Problems are Hard to Analyze

Accuracy problems can be difficult to identify and debug, and they can have serious consequences. Section 2 presented one such problem. While this problem was present in Stan, it is interesting that Stan's precursor, BUGS, which shares most of its modeling syntax and principles, computes the correct result. For a non-expert, it is often hard to figure out the reasons behind this discrepancy. Next, we provide some insights into how we analyzed this particular case.

We observed that the error is reproduced for any value of the parameters of Beta distribution, which is the prior for p. Stan produced warning messages stating that the random variable used for computing the beta logpdf in a particular step is negative but was expected to be positive. The Stan manual describes such messages as follows: *Warning messages arise in circumstances from which the underlying program can continue to operate* [109]. Stan often converges to the correct result despite such warnings, but in this case, it did not. When such warnings persist, Stan developers suggest "investigating the arithmetic stability of the Stan program" [109].

One way to address the accuracy problem is to change the model. Stan developers often recommend to manually bound the variables that have finite support [110]. For p in Figure 2a, we can set the bounds as follows: `real⟨lower=0, upper=1⟩p;` This makes Stan produce the correct output: 7.0. The origin of the problem lies in the way Stan does sampling. For any sampling statement of the form: `p ~ beta(a,b)`, Stan computes the log probability as: `target += beta_lpdf(p | a,b)` and updates the log density (logpdf) of the model. Beta distribution has a support of $(0, 1)$. If p is assigned a value outside this range, it causes logpdf to be undefined, which affects convergence. When the bounds are manually bound, Stan ensures that the parameter is in the valid range.

Such properties that are important for inference are not enforced. Stan's development has been influenced by "Folk Theorem" [96], which implies that in case of a wrong inference, the problem can be overcome by changing the model, and moreover the inference algorithms should not be made to work for various uncommon extreme program/data cases [96]. However, the "Folk Theorem" assumes that a developer has an intuition about the correct result, which may often not be the case as the PP systems are becoming mainstream. To help developers overcome such challenges, PP systems should provide additional information about the problems in the interaction between the model and the system. Recently, Stan developers proposed a "pedantic mode", as a way to diagnose various errors and bad modeling practices before running the inference [111], including range checks. We find this an interesting direction that can demonstrate the power of both probabilistic reasoning and static analysis, similar to lightweight static analyses in the traditional software development, e.g., [6].

Lesson Learned: Debugging accuracy problems requires not only domain knowledge but also a reasonable understanding of the PP system under test. The warning messages often provide hints if there is something wrong with the model. But the messages might not be informative enough to guide the user in fixing the model. This parallels the observations from compiler research on the importance of informative warnings for subsequent developer action [4, 5, 113].

Going forward, we note a promising application of static analysis to provide explicit hints about the model and its interaction with the inference algorithm without having to run the program. Like in compilers, they could provide *"useful warnings to alert developers to potentially problematic code fragments" and "suggestions to eliminate the warning"* [113] in the probabilistic setting. Tools like ProbFuzz have the potential to empirically discover the kinds of models that do not work well with a specific inference algorithm and inform such static analysis.

6.3 Fixing Bugs in PP systems is Non-Trivial

We found out that fixing the bugs, even the relatively straightforward dimension/boundary-value ones, is highly non-trivial and often involves changes to the design of the infrastructure (e.g., PyTorch and TensorFlow), that PP systems are built on.

As an example of a non-trivial problem, we reported a bug to Stan developers, which appears in some situations when the model is provided with an empty data array. In those cases the programs fails unexpectedly. The developers acknowledged the issue immediately, but even after an extensive discussion, the developers still have not been able to resolve the problem after several months.

In Edward, we submitted a pull request to ensure that the `n_samples` parameter of KLPQ inference was ≥ 0. The developers asked for the same fix to be made in several places in the KLQP inference: *"Cool! Can you also add this change to* `klqp.py` *for each* `initialize()` *method?"* We did as requested and our pull request was accepted and merged.

In Pyro, we identified that many distributions used from PyTorch do not have range checks. As we were discussing the potential fix with the developers, a PyTorch contributor independently started implementing their version of the fix. We discovered that the contributor's proposed fix had several bugs. Our tests that revealed bugs in the contributor's fix were driven by the failures that we had seen while running ProbFuzz-generated programs. Consequently, the contributor agreed to let us lead the fix, which has been approved for merging to the PyTorch repository.

Lesson Learned: Bugs in PP systems are not trivial to fix. Tests generated by ProbFuzz can help identify the causes of the problems. Moreover, automated testing can help discover incorrect and incomplete fixes.

6.4 Fixes Extend Across System Boundaries

As we were analyzing bugs and developing fixes, we found out that the failures that manifest in a PP system are often due to faults in the underlying infrastructure (PyTorch and TensorFlow). Therefore, some of the fixes we submitted were accepted by the developers of the underlying infrastructure. The accepted changes include a part of distribution checks in PyTorch plus many accompanying test cases and a fix to a bound for error reporting in TensorFlow.

We submitted our initial issues to the developers of Edward and Pyro, and they would typically direct us to propagate the fix to the underlying infrastructure. For Edward, the developer's response to our request to enable detailed checking of distribution ranges was *"That's an interesting suggestion...potentially useful utility. Can you raise this in TensorFlow?"* In Pyro, we submitted a pull request which added a check to prevent division by zero errors. Multiple Pyro developers responded and asked us to make the checks in PyTorch, so more people in the community will benefit: *"hello. thanks for the contribution! ... a more appropriate place... would be in pytorch... Thanks, this looks helpful...Thumbs up! I agree the PyTorch folks would appreciate better error checking, then a larger community could benefit from this fix."* The fix was accepted by PyTorch.

Lesson Learned: In PP systems, the errors and fixes can often extend across the boundaries of individual systems. ProbFuzz was effective in identifying such bugs since it analyzed and compared the end-to-end results of these composed systems.

7 DISCUSSION

Comparison with Traditional Testing Approaches. Inference bugs often require probabilistic reasoning and reasoning about accuracy, using, e.g., domain-specific oracles, metamorphic relations, or multiple implementations. As such, this category of bugs can be hard to catch using traditional testing techniques. Common techniques, such as coverage-based testing, would have problems because many of these bugs were caused by "faults of omission" [73]. Further, even bugs in covered code may require special values to manifest. Mutation testing of PP system code can potentially identify some bugs that result in program crashes or special values, but non-equivalent mutant survivals may indicate valid approximations rather than bugs [45], especially as tests in PP systems often only check whether the result lies in a loosely defined interval.

For the other bug categories, we give examples of previously unknown bugs that illustrate the advance of ProbFuzz over traditional testing approaches. For example, a dimension/boundary-value bug in Pyro manifested only when required parameters in two different functions were simultaneously out of acceptable ranges [20]. Conventional boundary-value analysis that targets one function at a time will not reveal this bug. As another example, in Edward, intermediate floating-point values produced by the SGLD inference algorithm led to `NaN` output when those values are "close enough" to the support bound [91]. Traditional boundary-value analysis may need to try many values near the bound to catch this bug. This bug remains open even after two workarounds that required advanced domain knowledge from the Edward developers.

A language/translation bug in Stan led to program crashes only on empty `int` arrays in the data, but not on empty `real` arrays [112]. Empty arrays are allowed in the data. The root-cause of the bug was that empty `int` arrays were implemented to be of data type `float`. Interestingly, the bug does not manifest in Stan itself, but in Stan's `PyStan` and `RStan` front-ends. Without the combination of domain knowledge on valid data elements and fuzzing, it will be difficult to catch such bugs with traditional testing techniques.

Scope. In our experiments, we used four templates, which focused on simple probabilistic models. Simple models can help developers understand potentially faulty executions and they were effective in finding bugs in the PP systems, but we did not aim for completeness of models in our evaluation. Going forward, PP system developers may also be interested in other common models that can be represented as templates in our language (e.g., hierarchical models, mixture models), and can be used to test various inference procedures, general or specialized for different model classes. However, ProbFuzz cannot generate arbitrary probabilistic programs, since its template language does not support while loops. Also, ProbFuzz is not suited for bugs that require precise analysis, e.g., [29, 103].

Threats to Validity. They include internal, external, and construct.

Internal. The results of our bug study depend on the set of PP systems and bugs we examined. We mitigated this risk by studying real bugs in three state-of-the-art PP systems. We may have wrongly characterized existing bugs as being inference-related. To mitigate this, two coauthors independently inspected the bugs and (when possible) the corresponding GitHub issues. We only mark a bug to be inference related if both coauthors eventually agree, thus achieving a conservative estimate of the number of inference-related bugs. We mitigate ProbFuzz implementation errors with unit testing. As differential testing may wrongly flag a program as potentially buggy, so we had multiple rounds of discussion among ourselves, and finally reported potential bugs to the PP system developers to make the final decision.

External. The results of the characterization study and ProbFuzz may not generalize to all PP systems. Certain aspects of our experimental design help to mitigate this risk. The three PP systems are being actively developed, well-tested, and adopted. We also demonstrated that ProbFuzz can reproduce existing bugs in each of the three bug categories across the PP systems.

Construct. ProbFuzz is designed to catch the categories of bugs identified by our study and may not find arbitrary bugs in PP systems. Discovery of these bugs is not exclusive to ProbFuzz. Other general and emerging testing techniques can, in principle, find some of the bugs identified in our evaluation.

8 RELATED WORK

Verification and analysis of probabilistic programs. There are various approaches for verification of probabilisitic programs, including probabilistic abstract interpretation [62, 66], symbolic execution [7, 32, 34, 59], probabilistic model checking [50], and other methods [16, 67, 70, 88]. Unlike these systems, ProbFuzz aims to find bugs in the systems on which probabilistic programs run, and not for debugging or analyzing probabilistic programs.

Program Generation for Compiler and System Testing. Several techniques have been proposed for generating programs that are used in system testing. These include techniques for generating programs to test compilers [3, 13, 14, 52–54, 56, 113, 123, 124] and to test refactoring engines and symbolic execution engines [18, 37, 49]. ProbFuzz also generates programs, but does so for a different class of systems: PP systems, which are characterized by various probabilistic constraints on how to construct programs and measure accuracy of the output (instead of binary correctness). One technical difference between ProbFuzz and earlier program-generation approaches is that ProbFuzz can generate programs in multiple languages—we currently generate Stan and Python from ProbFuzz, but more can be easily added. Lastly, ProbFuzz generates both programs and the data needed to run the programs, whereas all prior techniques generate only the programs (for compiler and system testing), or only the data (for testing programs).

Fuzzing. Researchers have previously proposed many fuzzing techniques [38, 39, 47, 60, 63, 65, 72, 74, 92, 93, 114, 123]. Grammar-based fuzzers [38, 63, 93, 123] encode knowledge about the structure of valid programs (more generally, inputs), but have no knowledge about the domain for which programs are typically written. The closest fuzzing approach that we found to ProbFuzz in terms of encoding domain knowledge is LangFuzz [47]. LangFuzz improves grammar-based fuzzing by first generating valid programs according to the grammar, and then mutating the programs based on knowledge about programs that previously caused invalid behavior. Therefore, LangFuzz incorporates domain knowledge in the form of historical invalid behaviors. In contrast to LangFuzz, the generation of programs by ProbFuzz already incorporates domain knowledge, without needing to perform any mutation or consider history.

Differential and Metamorphic Testing. Differential testing [43, 64, 121], or multiple-implementation testing [15, 55, 94, 116] use multiple implementations as oracles to find programs that can likely reveal bugs in PP system. ProbFuzz uses such approach in its Program Checker. Many problems in machine learning do not have a reference result, known as the 'no oracle' problem [68]. One solution to this problem is metamorphic testing [11, 12, 42, 58, 68, 69, 89, 122, 125], where *metamorphic relations* between the inputs and outputs of a program (or function) are leveraged to find inputs which cause outputs to diverge. Because metamorphic relations are hard to design, Srisakaokul et al. [94] recently proposed multiple-implementation testing of supervised machine-learning algorithms to find bugs. Implementations which classify differently from the majority are considered potentially buggy. Our differential testing of multiple inference algorithms is similar to [94].

9 CONCLUSION

We presented the first study of existing bugs in probabilistic programming systems (PP systems), and proposed ProbFuzz for testing for such bugs. ProbFuzz generates probabilistic programs from a user-specified template for three PP systems: Edward, Pyro, and Stan. Our study of historical bugs in Edward, Pyro, and Stan showed that numerical bugs, accuracy bugs and dimensional and boundary-value bugs form the majority of bugs. We demonstrated the ease of extending ProbFuzz by supporting several PP system versions in our study, and the applicability of ProbFuzz by showing that it can find existing bugs in the aforementioned categories in all three PP systems. ProbFuzz is already providing practical value: we reported 67 previously unknown bugs that we found by running ProbFuzz on recent versions of the three PP systems. We created pull requests with fixes for many of these bugs, 51 of which have been accepted by the developers. We believe that ProbFuzz opens a new line of research on testing probabilistic programming systems.

10 ACKNOWLEDGEMENTS

This work was funded in part by NSF Grants No. CCF-1703637, CCF-1629431, and CCF-1421503. We would also like to thank Milos Gligoric, Darko Marinov, Martin Rinard, and the anonymous reviewers for their comments on the research presented in this paper.

REFERENCES

[1] Eric Atkinson and Michael Carbin. 2016. Towards correct-by-construction probabilistic inference. In *NIPS Workshop on Machine Learning Systems*.
[2] Adrian Bachmann, Christian Bird, Foyzur Rahman, Premkumar Devanbu, and Abraham Bernstein. 2010. The missing links: bugs and bug-fix commits. In *FSE*.
[3] Antoine Balestrat. 2018. CCG - random C Code Generator. https://github.com/Mrktn/ccg.
[4] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In *ICSE*.
[5] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *SANER*.
[6] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: using static analysis to find bugs in the real world. *CACM* 53, 2 (2010).
[7] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, and Corina S Păsăreanu. 2015. Iterative distribution-aware sampling for probabilistic symbolic execution. In *FSE*.
[8] Johannes Borgström, Andrew D Gordon, Michael Greenberg, James Margetson, and Jurgen Van Gael. 2011. Measure transformer semantics for Bayesian machine learning. In *ESOP*.
[9] Bob Carpenter. 2017. https://github.com/stan-dev/stan/issues/603.
[10] Bob Carpenter, Andrew Gelman, Matt Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Michael A Brubaker, Jiqiang Guo, Peter Li, Allen Riddell, et al. 2016. Stan: A probabilistic programming language. *JSTATSOFT* 20, 2 (2016).
[11] Tsong Y Chen, Shing C Cheung, and Siu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. CS Department, Hong Kong University of Science and Technology.
[12] Tsong Yueh Chen, Jianqiang Feng, and TH Tse. 2002. Metamorphic testing of programs on partial differential equations: a case study. In *COMPSAC*.
[13] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *PLDI*.
[14] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *PLDI*.
[15] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. 2010. WEBDIFF: Automated identification of cross-browser issues in web applications. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.
[16] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference Using Data Flow Analysis. In *FSE*.

[17] Gregory F Cooper. 1990. The computational complexity of probabilistic inference using Bayesian belief networks. *Artificial intelligence* 42, 2 (1990).
[18] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. 2007. Automated Testing of Refactoring Engines. In *ESEC/FSE*.
[19] Anthony Di Franco, Hui Guo, and Cindy Rubio-González. 2017. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 509–519.
[20] Domain Error in arguments 2018. https://github.com/uber/pyro/issues/875.
[21] Edward Commit 002a27e 2016. https://github.com/blei-lab/edward/commit/002a27e.
[22] Edward Commit 10118db 2016. https://github.com/blei-lab/edward/commit/10118db.
[23] Edward Commit 3616d41 2016. https://github.com/blei-lab/edward/commit/3616d41.
[24] Edward commit 3616d41 2016. https://github.com/blei-lab/edward/commit/3616d41.
[25] Edward commit 43d8a39 2016. https://github.com/blei-lab/edward/commit/43d8a39.
[26] Edward Commit 79f4193 2017. https://github.com/blei-lab/edward/commit/79f4193.
[27] Edward commit 79f4193 2017. https://github.com/blei-lab/edward/commit/79f4193.
[28] Edward Commit 972a9d9 2017. https://github.com/blei-lab/edward/commit/972a9d9.
[29] Edward Commit c9afcb1f 2017. https://github.com/blei-lab/edward/commit/c9afcb1.
[30] Edward Commit fe01657 2016. https://github.com/blei-lab/edward/commit/fe01657.
[31] EdwardWebPage 2018. Edward. http://edwardlib.org.
[32] Antonio Filieri, Corina S Păsăreanu, and Willem Visser. 2013. Reliability analysis in symbolic pathfinder. In *ICSE*.
[33] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *CAV*.
[34] Jaco Geldenhuys, Matthew B Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *ISSTA*.
[35] Andrew Gelman, Daniel Lee, and Jiqiang Guo. 2015. Stan A Probabilistic Programming Language for Bayesian Inference and Optimization. *Journal of Educational and Behavioral Statistics* (2015).
[36] Wally R Gilks, Andrew Thomas, and David J Spiegelhalter. 1994. A language and program for complex Bayesian modelling. *The Statistician* (1994).
[37] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. 2010. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 225–234.
[38] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based Whitebox Fuzzing. In *PLDI*.
[39] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing.. In *NDSS*.
[40] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).
[41] Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. 2014. Probabilistic programming. In *FoSE*.
[42] Arnaud Gotlieb and Bernard Botella. 2003. Automated metamorphic testing. In *COMPSAC*.
[43] Alex Groce, Gerard Holzmann, and Rajeev Joshi. 2007. Randomized differential testing as a prelude to formal verification. In *ICSE*.
[44] Roger B Grosse, Siddharth Ancha, and Daniel M Roy. 2016. Measuring the reliability of MCMC inference with bidirectional Monte Carlo. In *NIPS*.
[45] Farah Hariri, August Shi, Owolabi Legunsen, Milos Gligoric, Sarfraz Khurshid, and Sasa Misailovic. 2018. Approximate Transformations as Mutation Operators. In *ICST*.
[46] Shawn Hershey, Jeff Bernstein, Bill Bradley, Andrew Schweitzer, Noah Stein, Theo Weber, and Ben Vigoda. 2012. Accelerating inference: towards a full language, compiler and hardware stack. *arXiv preprint arXiv:1212.2991* (2012).
[47] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *USENIX Security*.
[48] Daniel Jackson and Craig A Damon. 1996. Elements of style: Analyzing a software design feature with a counterexample detector. *TSE* 22, 7 (1996).
[49] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov. 2009. Reducing the Costs of Bounded-Exhaustive Testing. In *FASE*.
[50] Nils Jansen, Christian Dehnert, Benjamin Lucien Kaminski, Joost-Pieter Katoen, and Lukas Westhofen. 2016. Bounded Model Checking for Probabilistic Programs. *ATVA* (2016).
[51] Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David M. Blei. 2015. Automatic Variational Inference in Stan. In *NIPS*.
[52] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*.
[53] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*.
[54] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized Stress-testing of Link-time Optimizers. In *ISSTA*.
[55] Nuo Li, JeeHyun Hwang, and Tao Xie. 2008. Multiple-implementation Testing for XACML Implementations. In *TAV-WEB*.
[56] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core Compiler Fuzzing. In *PLDI*.
[57] Daniel Liew, Daniel Schemmel, Cristian Cadar, Alastair F Donaldson, Rafael Zähl, and Klaus Wehrle. 2017. Floating-point symbolic execution: A case study in N-version programming. In *ASE*.
[58] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. 2014. How effectively does metamorphic testing alleviate the oracle problem? *TSE* 40, 1 (2014).
[59] Kasper Luckow, Corina S Păsăreanu, Matthew B Dwyer, Antonio Filieri, and Willem Visser. 2014. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *ASE*.
[60] Rupak Majumdar and Ru-Gang Xu. 2007. Directed Test Generation Using Symbolic Grammars. In *ASE*.
[61] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint 1404.0099* (2014).
[62] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. 2013. Dynamic enforcement of knowledge-based security policies using probabilistic abstract interpretation. *JCS* 21, 4 (2013).
[63] Peter M. Maurer. 1990. Generating test data with enhanced context-free grammars. *Ieee Software* 7, 4 (1990), 50–55.
[64] William M McKeeman. 1998. Differential testing for software. *DEC Digital Technical Journal* 10, 1 (1998).
[65] Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *CACM* 33, 12 (1990).
[66] David Monniaux. 2000. Abstract interpretation of probabilistic semantics. In *SAS*.
[67] Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic predicate transformers. *TOPLAS* 18, 3 (1996).
[68] Christian Murphy and Gail E Kaiser. 2010. *Improving the dependability of machine learning applications*. Technical Report. CS Department, Columbia University.
[69] Christian Murphy, Gail E Kaiser, Lifeng Hu, and Leon Wu. 2008. Properties of Machine Learning Applications for Use in Metamorphic Testing.. In *SEKE*.
[70] Chandrakana Nandi, Dan Grossman, Adrian Sampson, Todd Mytkowicz, and Kathryn S McKinley. 2017. Debugging probabilistic programs. In *MAPL*.
[71] Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. 2014. R2: An efficient MCMC sampler for probabilistic programs. In *AAAI*.
[72] Peter Oehlert. 2005. Violating Assumptions with Fuzzing. *IEEE Security and Privacy* 3, 2 (2005).
[73] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *ICSE*.
[74] Paul Purdom. 1972. A sentence generator for testing parsers. *BIT Numerical Mathematics* 12, 3 (1972).
[75] Pyro Commit 0f1d27e 2017. https://github.com/uber/pyro/commit/0f1d27e.
[76] Pyro commit 3671b06 2017. https://github.com/uber/pyro/commit/3671b06.
[77] Pyro Commit 6e26f5e 2017. https://github.com/uber/pyro/commit/6e26f5e.
[78] Pyro Commit 7b6cf58 2017. https://github.com/uber/pyro/commit/7b6cf58.
[79] Pyro commit 7b6cf58 2017. https://github.com/uber/pyro/commit/7b6cf58.
[80] Pyro Commit 8c14f36 2017. https://github.com/uber/pyro/commit/8c14f36.
[81] Pyro Commit b94f06a 2017. https://github.com/uber/pyro/commit/b94f06a.
[82] Pyro Commit f5a51fe 2017. https://github.com/uber/pyro/commit/f5a51f.
[83] Pyro Issue 303 2017. https://github.com/uber/pyro/issues/303.
[84] PyroWebPage 2018. Pyro. http://pyro.ai.
[85] PyTorchWebPage 2018. PyTorch. http://pytorch.org.
[86] Michael Rath, Jacob Rendall, Jin LC Guo, Jane Cleland-Huang, and Patrick Mäder. 2018. Traceability in the wild: automatically augmenting incomplete trace links. In *ICSE*.
[87] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H Bailey, Costin Iancu, and David Hough. 2013. Precimonious: Tuning assistant for floating-point precision. In *SC*.
[88] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static analysis for probabilistic programs: inferring whole program properties from finitely many paths. *ACM SIGPLAN Notices* 48, 6 (2013).
[89] Sergio Segura, Gordon Fraser, Ana B Sanchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on software engineering* 42, 9 (2016), 805–824.
[90] Daniel Selsam, Percy Liang, and David L Dill. 2017. Developing Bug-Free Machine Learning Systems With Formal Mathematics. *arXiv preprint*

arXiv:1706.08605 (2017).
[91] SGLD produces Nan in output 2018. https://github.com/blei-lab/edward/issues/859.
[92] Guoqiang Shu, Yating Hsu, and David Lee. 2008. Detecting Communication Protocol Security Flaws by Formal Fuzz Testing and Machine Learning. In *FORTE*.
[93] Emin Gün Sirer and Brian N. Bershad. 1999. Using Production Grammars in Software Testing. In *DSL*.
[94] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. 2018. Multiple-Implementation Testing of Supervised Learning Software. In *EDSMLS*.
[95] Stan 2018. http://mc-stan.org.
[96] Stan Best Practices 2018. https://github.com/stan-dev/stan/wiki/Stan-Best-Practices.
[97] Stan Commit 04fcb74 2013. https://github.com/stan-dev/stan/commit/04fcb74.
[98] Stan commit 04fcb74 2013. https://github.com/stan-dev/stan/commit/04fcb74.
[99] Stan Commit 2fc94d4 2017. https://github.com/stan-dev/stan/commit/2fc94d4.
[100] Stan Commit 40c8224 2016. https://github.com/stan-dev/stan/commit/40c8224.
[101] Stan commit 45a82fd 2015. https://github.com/stan-dev/stan/commit/45a82fd.
[102] Stan Commit 5224850 2015. https://github.com/stan-dev/stan/commit/5224850.
[103] Stan Commit 5845db97 2016. https://github.com/stan-dev/stan/commit/5845db9.
[104] Stan Commit 7a98bd2 2016. https://github.com/stan-dev/stan/commit/7a98bd2.
[105] Stan Commit 99289c85 2015. https://github.com/stan-dev/stan/commit/99289c85.
[106] Stan Commit ae423b2 2017. https://github.com/stan-dev/stan/commit/ae423b2.
[107] Stan Commit b8d5086 2013. https://github.com/stan-dev/stan/commit/b8d5086.
[108] Stan Issue #2178 2017. https://github.com/stan-dev/stan/issues/2178.
[109] Stan Language Manual. Appendix D. 2018. http://mc-stan.org/users/documentation/index.html.
[110] Stan Language Manual. Chapter 3.1. 2018. http://mc-stan.org/users/documentation/index.html.
[111] Stan Pedantic Mode 2018. https://github.com/stan-dev/stan/wiki/Stan-Parser-Pedantic-Mode.
[112] Stan throws error with empty array 2018. https://github.com/stan-dev/pystan/issues/437.
[113] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *ICSE*.
[114] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.
[115] Symmetric Mean Absolute Percentage Error. Armstrong 1985. http://www.forecastingprinciples.com/files/LRF-Ch13b.pdf.
[116] Kunal Taneja, Nuo Li, Madhuri R. Marri, Tao Xie, and Nikolai Tillmann. 2010. MiTV: Multiple-implementation Testing of User-input Validators for Web Applications. In *ASE*.
[117] TensorFlowWebPage 2018. TensorFlow. https://www.tensorflow.org.
[118] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. 2017. Deep probabilistic programming. In *ICLR*.
[119] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv* (2016).
[120] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A new approach to probabilistic programming inference. In *AISTATS*.
[121] Tao Xie, Kunal Taneja, Shreyas Kale, and Darko Marinov. 2007. Towards a framework for differential unit testing of object-oriented programs. In *AST*.
[122] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *JSS* 84, 4 (2011).
[123] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*.
[124] Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. In *PLDI*.
[125] Zhi Quan Zhou, DH Huang, TH Tse, Zongyuan Yang, Haitao Huang, and TY Chen. 2004. Metamorphic testing and its applications. In *ISFST*.