Debugging with Intelligence via Probabilistic Inference

Zhaogui Xu[1], Shiqing Ma[2], Xiangyu Zhang[2]*, Shuofei Zhu[1]and Baowen Xu[1]*
[1] State Key Laboratory of Novel Software Technology, Nanjing University, China
[2] Department of Computer Science, Purdue University, USA

ABSTRACT

We aim to debug a single failing execution without the assistance from other passing/failing runs. In our context, debugging is a process with substantial uncertainty – lots of decisions have to be made such as what variables shall be inspected first. To deal with such uncertainty, we propose to equip machines with human-like intelligence. Specifically, we develop a highly automated debugging technique that aims to couple human-like reasoning (e.g., dealing with uncertainty and fusing knowledge) with program semantics based analysis, to achieve benefits from the two and mitigate their limitations. We model debugging as a probabilistic inference problem, in which the likelihood of each executed statement instance and variable being correct/faulty is modeled by a random variable. Human knowledge, human-like reasoning rules and program semantics are modeled as conditional probability distributions, also called probabilistic constraints. Solving these constraints identifies the most likely faulty statements. Our results show that the technique is highly effective. It can precisely identify root causes for a set of real-world bugs in a very small number of interactions with developers, much smaller than a recent proposal that does not encode human intelligence. Our user study also confirms that it substantially improves human productivity.

CCS CONCEPTS

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

KEYWORDS

Debugging, Probabilistic Inference, Python

ACM Reference Format:
Zhaogui Xu, Shiqing Ma, Xiangyu Zhang, Shuofei Zhu and Baowen Xu. 2018. Debugging with Intelligence via Probabilistic Inference. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 11 pages.
https://doi.org/10.1145/3180155.3180237

1 INTRODUCTION

In this paper, we aim to tackle the traditional debugging problem – given a faulty program and a single failing run, identify the root cause of the failure. In our target scenario, an oracle to tell the intended behavior for each step of execution is not available. As such, debugging becomes an uncertain procedure, in which a lot of decisions have to be made, such as which statements are more likely faulty, which variables should be inspected first, and whether a function execution shall be stepped into. *Algorithmic debugging* techniques tend to avoid making decisions by conservatively including all the possibilities. For example in dynamic slicing, given a faulty output, all the executed statement instances that have contributed to the output are included in the slice. While these techniques can precisely model and reason about program semantics, they lack the capabilities of making appropriate predictions in the presence of uncertainty. The onus is hence on the developers to inspect the large volume of analysis results. In contrast, during *human debugging*, an experienced developer may not even start from the vicinity of the faulty output. Instead, she may speculate some places in the middle of execution that contain states critical to the failure and inspect the corresponding variables. She decides if a variable has a faulty value based on her experience and domain knowledge. In many cases, she can quickly identify the root cause because she excels at collecting and fusing debugging hints to make the correct decisions. On one hand, many of these hints are highly uncertain (e.g., the variable name correlations between the faulty output variable and the root cause) and hence can hardly be leveraged by algorithmic debugging techniques. On the other hand, humans do not have the capacity and stamina to reason about the low level program semantics at a scale close to machines.

Human feedback driven debugging was hence proposed to integrate human reasoning and machine analysis [24, 26, 32]. For instance, in a very recent proposal of interactive slicing [24], the tool reports statement instances in a dynamic slice to the developer, one at a time based on their distance to the faulty output. The developer can indicate if a reported instance is faulty; and if not, what operands of the instance are faulty. The tool then recomputes the dynamic slice from the faulty operands, precluding statement instances that led to the correct operands (in the instance). However, in most existing techniques including [24], the coupling between machines and humans is very primitive: the machine analysis is incapable of handling uncertainty and humans still have to make all the decisions. As a result, they suffer from limitations such as excessive requests for human intervention and false positives/negatives due to human mistakes.

We develop a technique that allows machines to take over a large part of the human reasoning of uncertainty, and couple such uncertainty reasoning with precise modeling of low level program semantics such that we can achieve the benefits of both human and machine reasonings and mitigate their limitations. In particular, we model debugging as a probabilistic inference procedure. Random variables are introduced to denote the likelihood of statement instances and

* Corresponding authors.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180237

variables being correct/faulty. The semantics of each executed statement is modeled as a conditional probability distribution, which is also called *probabilistic constraint* in this paper (e.g., given "x=y", if x is likely correct, then y is likely correct). The outputs are modeled as observations, e.g., the correct outputs are associated with probability 1.0 (of being correct) and the faulty outputs with probability 0.0. Human domain knowledge and feedback are also encoded as probabilistic constraints (e.g., the likelihood of being buggy for a variable with a name correlated to the name of the faulty output variable is higher than others). These constraints can be resolved by a probabilistic inference engine [1]. The inference procedure is similar to solving SMT/SAT constraints. The difference lies in that our inference results are posterior marginal probabilities (instead of satisfying value assignments) that indicate the likelihood of each statement instance and variable being correct/faulty (given the observations and the human feedback). The procedure is analogous to applying forces to an elastic mesh, in which the observations and human feedback are analogous to the forces and the correlations between statements/variables derived from program semantics are analogous to the mesh. When the mesh finally stabilizes, the state of each node on the mesh reflects the effect of the joint force.

Compared to the aforementioned existing feedback driven debugging, (1) developers' intervention/feedback is substantially reduced as the technique can perform human-like reasoning in dealing with uncertainty; (2) our technique allows developers to be uncertain about their feedback and human mistakes can be diluted and eventually over-powered by other evidences; and (3) it has the full capacity of machine analysis by faithfully encoding program semantics.

Our contributions are summarized as follows.

- We propose the idea of modeling debugging (a single failing run) as a probabilistic inference problem such that our technique can automatically perform both human-like reasoning that features uncertainty handling and program semantics based reasoning that features precision.
- We devise a comprehensive set of rules to encode both program semantics, observations, domain knowledge, and human feedback as conditional probability distributions.
- We develop a prototype for debugging Python programs. Our evaluation on real world bugs shows that on average our tool can locate a root cause in 3 steps of interaction with the developer for large projects, and 5 steps for smaller programs from stackoverflow. In contrast, [24] requires more than 10 steps and may miss the root causes. Our user study shows that our tool can reduce debugging time by 34% on average. The analysis time is within seconds for most cases.

2 MOTIVATION

Motivating Example. Fig. 1 shows a simplified code snippet from a popular Python HTTP library `Requests` [2]. Line 18 invokes function `path_url()` defined in lines 1-12. Inside the function, the program first splits the input `url` into `path` and `query` (line 2). With the failure inducing input, the resulted values of `path` and `query` are `"/t%20c"` and `"x=1"`, respectively. Line 3 encodes the path segment of the url by replacing special characters with their encodings (e.g., space to '%20' and '%' to '%25') to translate it to the standard HTML

```
1:  def path_url(url):
2:    path, query = urlsplit(url)
3:    path = encode(path)
4:    if not path:
5:      path = "/"
6:    purl = []
7:    purl.append(path)
8:    if query:
9:      purl.append("?")
10:     purl.append(query)
11:   purl_str = make_str(purl)
12:   return purl_str

13: def make_str(lst):
14:   s = ""
15:   for i in lst:
16:     s = s + i
17:   return s

18: url=path_url("http://x.org/t%20c?x=1")
19: print url
```

Figure 1: Motivating example.

URI format. Hence, the encoded `path` has the value `"/t%2520c"` (after replacing '%' with '%25'). Lines 4-5 validate that `path` is not empty. Line 6 appends `path` to a list `purl`. Lines 8-10 further append a separator symbol '?' and `query` to the list. Line 11 calls the function `make_str()` to transform `purl` to a string. Inside `make_str()`, lines 13-17 concatenate each item in `purl` and return the final value `"/t%2520c?x=1"`. This code snippet is buggy at line 3 because the path is double encoded, and the correct version should check if it has already been encoded before executing line 3. The expected output should be `"t%20c?x=1"` with '%20' the encoding of space.

Existing Methods. In manual debugging, developers use pdb [3] and PyCharm [4] debugger (i.e., the Python version of gdb) to set breakpoints at places that are considered suspicious and inspect variable values at those points. In this procedure, developers have to make a lot of decisions to deal with the inherent uncertainty. Many of the decisions are made based on their experience. For example, the library function `append()` is unlikely to be buggy so that the developers do not need to step into its execution. However, the effectiveness of manual debugging heavily hinges on the experience of developers. Moreover, humans have limited capacity of reasoning about low level program semantics. For example, it is difficult for humans to use debuggers to follow a lengthy data flow path.

Many automated debugging techniques, such as delta debugging [8–11] and fault localization [15–18, 20], are highly effective. But they often require a passing run that closely resembles the failing run, or a large set of passing and failing runs, to serve as the reference to suppress uncertainty. However in practice, reference run(s) with high quality may not be available.

Dynamic slicing identifies a subset of executed statements that contributed to an observed faulty output based on program dependencies. It makes very conservative assumptions in dealing with uncertainty. For instance, line 16 concatenates two substrings – right-hand-side (rhs) `s` and `i`, to produce the left-hand-side (lhs) `s`. If the lhs `s` is observed faulty, it assumes both the rhs `s` and `i` are potentially faulty and includes both in the slice. In our example, the dynamic slice of the printed value of variable `url` at line 19 includes all the statements presented in this example, and also the bodies of functions `urlsplit()` and `encode()` that are omitted. It hence requires a lot of manual efforts to go through statements in the slice in order to identify the root cause.

Recently, an interactive approach [24] was proposed. It gradually presents statement instances in a dynamic slice to the developer, who provides feedback such as whether an instance itself is faulty. If not, the developer shall indicate which operand of the instance is faulty. The technique then computes a new slice starting from the faulty operand. It essentially prunes part of the original slice related to the correct operand(s). Consider our example, it first computes the slice of line 19. The slice includes all the executed statement instances. It first presents lines 19, 18 and 17 to the developer, one at a time. The developer indicates that those statements themselves are not faulty but rather their operands are faulty. It then presents line 16 to the developer, who indicates that the rhs `s` is faulty but `i` is not. As such, the technique computes a new slice on `s`, precluding `i` and its dependencies. Although the technique does improve over slicing, its integration between machines and humans is primitive. The algorithm is fully deterministic and does not handle uncertainty, and the developer merely serves as an oracle. As such, the number of human interactions required can be large. In our example, it requires 7 interactions until the root cause is identified. Moreover, it does not handle/tolerate human mistakes. For example, if the developer mistakenly determines `i` at line 16 is faulty and the rhs `s` is correct, the root cause can never be reached.

Our Idea. We propose an automated technique that couples human-like reasoning (e.g., handling uncertainty, leveraging domain knowledge, and fusing debugging hints from various sources) with precise low level program semantics based analysis. As such, a large part of the human workload can be shifted to machines. The idea is to formulate debugging as a probabilistic inference problem. We introduce *random variables* to denote the likelihood of individual statement instances and variables being correct/faulty, encode program semantics, human reasoning rules as *probabilistic constraints*, and inputs, (faulty) outputs, human feedback as *observations* that are denoted as *prior probabilities* (i.e., probabilities before inference). The prior probabilities and constraints are fed to the probabilistic inference engine to compute the *posterior probabilities* of individual statement instances and variables being correct/faulty. The posterior probability of a variable is often different from its prior probability, denoting a better assessment after fusing information from other related variables.

Next, we will walk through the example to intuitively explain how our technique identifies the root cause. Before inference, `url` at line 19 is associated with a constant prior probability $LOW = 0.05$ to indicate it is faulty and `url` at line 1 is associated with $HIGH = 0.95$ to indicate that it is correct. It is standard in probabilistic inference not to use 0.0 or 1.0, but rather values very close to them [50]. Since we have no observations on other program variables, their prior probabilities are set to a constant $UNCERTAIN = 0.5$. The inference engine takes the prior probabilities and the probabilistic constraints derived from program semantics, and computes the posterior probabilities. Next, we present some of the computed posterior probabilities and explain their intuition.

From the program semantics constraints, `url` is faulty at line 19 suggests that `url` at 18 and `purl_str` at line 12 are *likely* faulty, with posterior probabilities (of being correct) computed as 0.0441 and 0.0832 after inference. The procedure of computing posterior probabilities from prior probabilities will be explained in later sections. The posterior probabilities model the following. There are two possible reasons leading to the faulty value of `url`. One is the executed statement at line 12 is buggy (but the operand `purl_str` is correct) and the other is the value of operand `purl_str` is faulty. Since there is no other observation indicating the simple return statement is faulty, the computed posterior probabilities indicate that it is more likely that `purl_str` has an incorrect value. Note that a variable being faulty does not mean our tool reports it to the developer as all the variables along the failure propagation path (i.e., the dependence path from the root cause to the faulty output) are faulty but we are interested in the root cause, which is a statement instance. Leveraging data flow constraints, our tool further infers that variable `purl_str` being likely faulty at line 12 entails that `s` at line 17 is likely faulty and hence `s` at line 16 likely faulty (with probability being 0.1176). Line 16 is executed three times, producing three `s` values, namely, `"/t%2520c"`, `"/t%2520c?"` and `"/t%2520c?x=1"`. When comparing the generated output with the expected output, our tool also marked the correctness of individual bytes. Hence, at this point, it can factor in the observation that the output substring `"20c?x=1"` created by the last two instances of line 16 is correct. This suggests that the root cause is unlikely at line 16, reflected by the computed posterior probability 0.6902. Instead, the first instance of `i` at line 16 is likely faulty.

Our tool also encodes various human domain knowledge as probabilistic constraints. For example, the name of the faulty output variable and the name of a function that contains the root cause tend to have some correlation in terms of natural language semantics. As such, function `path_url()` is considered more relevant than function `make_str()`, reflected by larger *prior probabilities* for statements in the former. The first instance of `i` at line 16 being faulty suggests that the resulted `purl` at line 7 (inside `path_url()`) is likely faulty (i.e., posterior probability 0.2892 being correct). Again, there are two possible sources that lead to the faulty state of `purl` at 7: (1) the `append()` function is faulty and (2) `path` is faulty. Our tool has encoded the domain knowledge that an external library function is less likely faulty than user code. This leads to (2) out-weighting (1) during inference. Note that it is not a problem for our tool even if the heuristics do not apply (e.g., the root cause does reside in library instead of user code) as our technique is probability based. The effect of a heuristic can be out-weighted by evidences collected from other sources.

Variable `path` at line 7 being faulty suggests either (A) the rhs `path` at line 3 is faulty or (B) line 3 itself is faulty. The observation that input `url` at line 1 being correct (as it is from the input) leads to `path` and `query` at line 2 are likely correct because they are identical to parts of `url`. This further suggests the rhs `path` at line 3 is likely correct. As such, (B) is likely true and hence line 3 is the statement instance with the lowest posterior probability (0.1075) being correct. Note that statement instance probabilities are different from variable probabilities as the former indicates where the root cause is. *Although we describe the procedure step-by-step, our tool encodes everything as prior probabilities and probabilistic constraints that are automatically resolved. In other words, the entire aforementioned procedure is conducted internally by our tool and invisible to the developer.* Also observe that the inference procedure is bidirectional, e.g., the backward reasoning from the output at line 19 and the forward reasoning from the input at line 1. It is iterative as well and terminates when a fixed point is reached. *These*

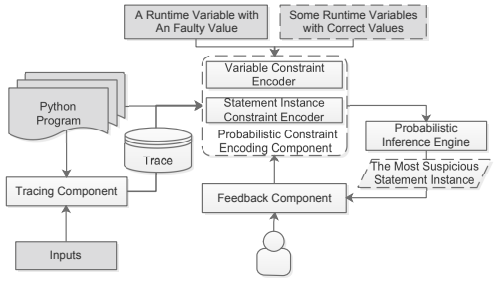

Figure 2: System Framework.

features distinguish our technique from iterative slicing (e.g., [24]) or techniques that present a dynamic slice to the developer through interactions (e.g., [32]), as slicing does not conduct any bidirectional or fixed-point (probabilistic) reasoning, but rather traverses program dependencies. The statement instance with the largest probability to be faulty is reported to the developer. If the developer disagrees, she can further provide her assessment about the *likelihood* that the operands in the reported statement instance are faulty. The feedback is encoded as a new observation, i.e., new prior probabilities that can be propagated backward and forward, and another round of inference is performed to identify the new root cause candidate. Note that human mistakes in feedback can be tolerated as they do not lead to the exclusion of the root cause but rather just a few more extra rounds before locating the root cause. This faithfully mimics how human mistakes are tolerated in real world manual debugging. In our example, our tool correctly reports the true positive in the first round, which is much more effective than [24].

3 OVERVIEW

In this section, we give an overview of our system.

Framework. Fig. 2 presents the system architecture of our tool. It consists of four components: the tracing component, the probabilistic constraint encoding component, the probabilistic inference engine and the feedback component. The tracing component takes the buggy program and a test input and generates the trace of a failing execution. The probabilistic constraint encoding component automatically encodes prior probabilities and constraints from various resources. It consists of two subcomponents: *variable correctness/faultiness probabilistic constraint encoder* and *statement instance correctness/faultiness encoder*. The former generates prior probabilities to indicate initial observations of variable correctness/-faultiness. Such prior probabilities have three possible constant values: $HIGH = 0.95$ to denote a variable having a correct value (e.g., an input variable or an output variable holding a correct value), $LOW = 1 - HIGH = 0.05$ to denote a variable having a faulty value (e.g., a faulty output variable), and $UNCERTAIN = 0.5$ to denote we have no observation/prior-knowledge about a variable (most variables fall into this category). It is standard to use these constant values as prior probabilities [50]. We will study the effect of having different $HIGH$ configurations in Section 6. Besides prior probabilities, it also generates constraints that correlate probabilities according to program semantics (e.g., $x = y$ dictates the strong correlation between x and y's probabilities). The inference engine takes the prior probabilities and the constraints, performs probability inference to compute posterior probabilities. The inference procedure can be intuitively considered as a process of fusing hints

$x \in RuntimeVariableSet \quad inst \in ExecutedStatementInstanceSet$
$\mathcal{P}(x/inst)$ asserts a variable x or a statement instance $inst$ is correct.
$\mathcal{S}(inst)$ asserts a statement instance $inst$ is correct from the program structure.
$\mathcal{N}(inst)$ asserts a statement instance $inst$ is correct from the naming convention.
$C : a \xrightarrow{p} b$ represents a probabilistic constraint denoting predicate a has a *propagation probability* p implying predicate b.

Figure 3: Basic Definitions.

from various observations (i.e., $HIGH/LOW$ prior probabilities) through the propagation channels dictated by the constraints. At the end, many variables that had $UNCERTAIN$ prior probabilities now have posterior probabilities different from $UNCERTAIN$, indicating their likelihood of being correct/faulty.

The second subcomponent, the statement instance encoder, automatically generates constraints to infer if individual statement instances are correct/faulty from variable probabilities and domain knowledge. The earlier inference results of variable correctness/-faultiness are provided as prior probabilities to the statement instance inference procedure. Eventually, the statement instances are ranked by their posterior probabilities. The one with the largest probability of being faulty is reported as the root cause candidate. If the developer disagrees, she provides feedback to indicate whether the reported statement instance contains faulty variable(s), which may trigger another round of encoding and inference. The process is iterative till the root cause is identified.

Motivation Example Walk-through. We simplify the execution trace of the motivating example in Fig. 1 as the following to illustrate the workflow of our technique.

e_1: url_2 = url_1 (L1)	e_4: s_1 = $path_1$ + "?" (L16)
e_2: p_1 = url_2[0] (L2)	e_5: s_2 = s_1 + $query_1$ (L16)
e_3: $path_1$ = m_1[p_1] (L3)	e_6: url_3 = s_2 (L17)

Table 1: Simplified Trace of the Motivating Example in Fig. 1.

All the runtime variables in the trace are transformed to the *Static Single Assignment* (SSA) form so that each of them is defined exactly once. The subscript of a variable is used to identify a specific instance of the variable. The line numbers of these trace events are also shown on their right (e.g., L2 means line 2). The value of variable url_3 is faulty whereas url_1 and $query_1$ are correct.

Phase I: Inferring Variable Correctness/Faultiness Probabilities. With the execution trace, our encoding component first performs dynamic slicing from the faulty output variable(s) and then encodes the semantics of each event in the slice. With the simplified trace in Table 1, the faulty output is url_3, whose dynamic slice contains all the events. Take event e_5 as an example. Given the basic definitions shown in Fig. 3, the variable constraints are encoded as follows.

$$\mathcal{P}(s_1) \wedge \mathcal{P}(query_1) \xrightarrow{0.95} \mathcal{P}(s_2) \quad (1)$$
$$\mathcal{P}(s_2) \wedge \mathcal{P}(query_1) \xrightarrow{0.95} \mathcal{P}(s_1) \quad (2)$$
$$\mathcal{P}(s_1) \wedge \mathcal{P}(s_2) \xrightarrow{0.95} \mathcal{P}(query_1) \quad (3)$$

Since we do not know which step executes a faulty statement in this phase, we initially assume every executed statement instance is likely correct. Therefore, given the assumption that the addition statement is correct in event e_5, constraint (1) represents if both operands are correct, the produced value is also correct with a high probability ($HIGH = 0.95$). This probability is associated with the constraint (instead of a variable) and called the *propagation probability*. Intuitively, it can be considered as an information flow throttle

during inference, controlling how much s_2's probability is affected by those of s_1 and $query_1$. Note that the power of probabilistic inference lies in automatically correcting *initial beliefs/assumptions* by fusing information. For instance, observations about s_2 or values computed from s_2 (in other statement instances) would allow the engine to adjust the probability of s_2 in e_5 and potentially the belief about the correctness of e_5.

Constraints (2) and (3) represent if the result as well as either one of the two operands are correct, the other operand is likely correct. Intuitively, since it is a one-to-one mapping between the operands and the result value for an addition, the propagation probability is *HIGH*. The propagation probability is lower for other statements that denote many-to-one mapping. For instance, for the statement executed at event e_3, if we know both $path_1$ and m_1 are correct, the likelihood of the index p_1 being correct is dependent on if there exist multiple indexes whose corresponding array values are the same as the value of $path_1$. If there are no other array elements that have the value of $path_1$, p_1 is correct. If there are many such elements, we essentially get no hints about p_1's correctness. The detailed computation rules of the propagation probability for each kind of statement are presented in Section 5.1. Besides encoding constraints, our tool also encodes prior probabilities (for inputs and outputs). For example, we have the following prior probabilities in the parentheses for the trace in Table 1.

$$\begin{aligned} &\mathcal{P}(url_3) = 1\ (0.05) \quad \mathcal{P}(url_1) = 1\ (0.95) \\ &\mathcal{P}(query_1) = 1\ (0.95) \end{aligned} \tag{4}$$

We send the encodings to a probabilistic inference engine to compute the posterior probability of each predicate $\mathcal{P}(x)$ being true. During probabilistic inference, a random variable is associated with each predicate to indicate the likelihood of the predicate being true. The inference engine transforms constraints and prior probabilities to a *factor graph* and then uses the *Sum-Product* [30] algorithm to compute the posterior probabilities. We will disclose more details about probabilistic inference in the next section. For this example, the posterior probabilities of predicates $\mathcal{P}(s_1)$, $\mathcal{P}(query_1)$ and $\mathcal{P}(s_2)$ being true are 0.1302, 0.9197 and 0.0741, respectively.

Phase II: Inferring Statement Instance Correctness/Faultiness Probabilities. In this phase, we leverage the variable probabilities (from Phase I) and domain knowledge to determine the likelihood of each executed statement instance being correct/faulty. Particularly, we generate three kinds of constraints. First, we generate constraints correlating variable probabilities and statement instance probabilities. For event e_5, we generate the following constraints.

$$\mathcal{P}_{e_5}(s_2) \wedge (\mathcal{P}_{e_5}(s_1) \wedge \mathcal{P}_{e_5}(query_1)) \xrightarrow{0.95} \mathcal{P}(inst_{e_5}) \tag{5}$$

$$\neg\mathcal{P}_{e_5}(s_2) \wedge (\mathcal{P}_{e_5}(s_1) \wedge \mathcal{P}_{e_5}(query_1)) \xrightarrow{0.95} \neg\mathcal{P}(inst_{e_5}) \tag{6}$$

$$\neg\mathcal{P}_{e_5}(s_2) \wedge (\neg\mathcal{P}_{e_5}(s_1) \vee \neg\mathcal{P}_{e_5}(query_1)) \xrightarrow{0.95} \mathcal{P}(inst_{e_5}) \tag{7}$$

Intuitively, constraint (5) represents if all the involved values at event e_5 are correct, the addition operation is likely correct. Constraint (6) represents if the resulted value is faulty and both operands are correct, the addition operation is likely faulty. Constraint (7) denotes if the resulted value is faulty and at least one operand is faulty, the addition operation is likely correct. According to constraint (7), since the probabilities of both $\mathcal{P}(s_2)$ (0.0741) and $\mathcal{P}(s_1)$ (0.1302) are low, the statement instance e_5 is likely correct and the root cause is likely before e_5.

Second, it generates constraints from program structure. In practice, programs often have modular design to achieve functional coherence. Therefore, if a function includes a large number of statements that are executed and included in the slice, the function is likely to include the faulty statement (i.e., root cause). In our example, more statements from function `path_url()` are included in the slice than from `make_str()`, hence statement instances in the former function are given priority over those in the latter. The constraint related to e_5 is shown in the following.

$$\mathcal{S}(inst_{e_5}) \xleftrightarrow{0.95} \mathcal{P}(inst_{e_5}) \quad \mathcal{S}(inst_{e_5}) = 1\ (0.70) \tag{8}$$

Predicate $\mathcal{S}(inst_{e_5})$ and its prior probability 0.70 represent the prediction of e_5 being correct based on its structure. Here, since e_5 is inside `make_str()`, its prior probability (of correctness) is higher than those in `path_url()`. The prior probability 0.70 is derived through program analysis (see Section 5.2). During inference, it adds weight to the posterior probability of $\mathcal{P}(inst_{e_5})$.

Third, our technique generates constraints from the naming convention. We assume that function names and variable names, including those of the observed faulty variables, follow certain naming conventions and these names suggest functionalities to some extent. If two variable names are correlated in the natural language perspective, their functionalities are possibly related too. For our example, the function name `"path_url"` is more similar to the name of the observed faulty variable `"url"` than the function name `"make_str"`. It suggests that statement instances in function `path_url()` have stronger correlations with the failure than those in function `make_str()` and hence shall be given higher priority. For example, the naming constraints of e_3 are shown as follows.

$$\mathcal{N}(inst_{e_3}) \xleftrightarrow{0.95} \mathcal{P}(inst_{e_3}) \quad \mathcal{N}(inst_{e_3}) = 1\ (0.275) \tag{9}$$

Predicate $\mathcal{N}(inst_{e_3})$ denotes if e_3 is predicted to be fault related by naming convention. The prior probability 0.275 is derived from NLP analysis (see Section 5.2).

Finally, we send these constraints and prior probabilities again to the inference engine to compute the posterior probabilities of statement instances. We report the most likely faulty instance. In this example, event e_3 has the smallest probability of being correct. It is indeed the true root cause.

4 PROBABILISTIC INFERENCE

In this section, we illustrate how probabilistic inference is conducted. First of all, we present some basic notations. We denote each involved predicate $\mathcal{P}$ by an individual random variable x. Given a set of probabilistic constraints C_1, C_2, ..., and C_m, we use a set of corresponding probabilistic functions f_1, f_2, ..., and f_m to describe the valuation of these constraints. Formally, a probabilistic function f_i can be presented as the following.

$$f_i(x_1, x_2, ..., x_k) = \begin{cases} p & \text{if the constraint } C_i \text{ is true.} \\ 1-p & \text{otherwise} \end{cases} \tag{10}$$

where $x_1, x_2, ..., x_k$ denote the random variables associated with the constraint C_i and p represents the prior probability of the constraint yielding true. Probabilistic inference is essentially to satisfy all the

Table 2: Boolean Constraints with Probabilities.

x_1	x_2	x_3	$f_1(x_1, x_2, x_3)$	$f_2(x_1, x_2, x_3)$	$f_3(x_2)$	$f_4(x_3)$
0	0	0	0.95	0.95	0.05	0.95
0	0	1	0.95	0.95	0.05	0.05
0	1	0	0.95	0.95	0.95	0.95
0	1	1	0.95	0.05	0.95	0.05
1	0	0	0.95	0.95	0.05	0.95
1	0	1	0.95	0.95	0.05	0.05
1	1	0	0.05	0.95	0.95	0.95
1	1	1	0.95	0.95	0.95	0.05

constraints. In this context, the conjunction of the constraints can be denoted as the product of all the corresponding probabilistic functions, as shown in the following.

$$f(x_1, x_2, ..., x_n) = f_1 \times f_2 \times ... \times f_m \tag{11}$$

The joint probability function [30], which is essentially the normalized version of $f(x_1, x_2, ..., x_n)$, is defined as follows.

$$p(x_1, x_2, ..., x_n) = \frac{f_1 \times f_2 \times ... \times f_m}{\sum_{x_1,...,x_n} (f_1 \times f_2 \times ... \times f_m)} \tag{12}$$

The posterior (marginal) probability of x_i, denoted as $p(x_i)$, is the sum over all variables other than x_i [30].

$$p(x_i) = \sum_{x_1} \sum_{x_2} ... \sum_{x_{i-1}} \sum_{x_{i+1}} ... \sum_{x_n} p(x_1, x_2, ..., x_n) \tag{13}$$

Example. We use the example in Section 3 to illustrate more details. For simplicity, we only consider the inference for constraints (1) and (2) in phase I for the event e_5. Let random variables x_1, x_2 and x_3 denote the predicate $\mathcal{P}(s_1)$, $\mathcal{P}(query_1)$ and $\mathcal{P}(s_2)$, respectively. Hence, we have the following formula for constraints (1) and (2).

$$C_1 : x_1 \wedge x_2 \xrightarrow{0.95} x_3 \quad C_2 : x_3 \wedge x_2 \xrightarrow{0.95} x_1 \tag{14}$$

We assume the prior probabilities of x_2 and x_3 are 0.05 and 0.95, respectively. They are denoted as follows.

$$C_3 : x_2 = 1\ (0.95) \qquad C_4 : x_3 = 1\ (0.05) \tag{15}$$

We then transform each constraint to a probability function, e.g., the probability function f_1 for constraint C_1 is presented as follows.

$$f_1(x_1, x_2, x_3) = \begin{cases} 0.95 & \text{if } (x_1 \wedge x_2 \rightarrow x_3) = 1 \\ 0.05 & \text{otherwise} \end{cases} \tag{16}$$

Others are transformed similarly. Table 2 presents the values of the probability functions. Assume we want to compute the posterior marginal probability $p(x_1 = 1)$, which means the probability of s_1 being correct. The computation is as follows.

$$\begin{aligned} p(x_1 = 1) &= \frac{\sum_{x_2,x_3} f_1(1, x_2, x_3) \times f_2(1, x_2, x_3) \times f_3(x_2) \times f_4(x_3)}{\sum_{x_1,x_2,x_3} f_1(x_1, x_2, x_3) \times f_2(x_1, x_2, x_3) \times f_3(x_2) \times f_4(x_3)} \\ &= \frac{0.95 \times 0.95 \times 0.05 \times 0.95 + \ldots + 0.95 \times 0.95 \times 0.95 \times 0.05}{0.95 \times 0.95 \times 0.05 \times 0.95 + \ldots + 0.95 \times 0.95 \times 0.95 \times 0.05} \\ &= \frac{0.1309}{0.9927} = 0.1319 \end{aligned} \tag{17}$$

It is the sum of the product of valuations with $x_1 = 1$ (e.g., the first item $0.95 \times 0.95 \times 0.05 \times 0.95$ is the product of $f_1(x_1 = 1, x_2 = 0, x_3 = 0) = 0.95$, $f_2(1, 0, 0) = 0.95$, $f_3(1, 0, 0) = 0.05$ and $f_4(1, 0, 0) = 0.95$) divided by the sum of the product of all valuations.

Implementation. In practice, the computation of posterior marginal probabilities is very expensive. In our implementation, we represent all the probabilistic constraints with a graphical model called *factor graph* [30], which supports efficient computation. We compute marginal probabilities based on the *Sum-Product* algorithm [30]. The algorithm is essentially a procedure of message passing (also called *belief propagation*), in which probabilities are only propagated between adjacent nodes. In a message passing round, each node first updates its probability by integrating all the messages it receives and then sends the updated probability to its downstream receivers. The algorithm is iterative and terminates when the probabilities of all nodes converge. Our implementation is based upon *libDAI* [1], a widely used open-sourced probabilistic graphical model library.

A. Variable Constraints

$$\mathcal{P}(pred) \wedge \bigwedge_{u \in Uses(e)} \mathcal{P}(u) \xrightarrow{HIGH} \mathcal{P}(def) \quad [1]$$

foreach $u \in Uses(e) \Rightarrow$

$$\mathcal{P}(pred) \wedge \mathcal{P}(def) \wedge \bigwedge_{x \in Uses(e)/u} \mathcal{P}(x) \xrightarrow{p_u} \mathcal{P}(u) \quad [2]$$

$$\mathcal{P}(def) \wedge \bigwedge_{x \in Uses(e)} \mathcal{P}(x) \xrightarrow{HIGH} \mathcal{P}(pred) \quad [3]$$

B. Computation Rules of p_u in [2]

$$\frac{s : z = v \mid x \mid x\ bop\ y \qquad bop \in \{+, -, \times, /\}}{p_u = HIGH} \quad [1TO1]$$

$$\frac{s : z = x \bmod y}{p_x = UNCERTAIN} \quad [MOD]$$

$$\frac{s : y = x.f \quad Values(y) = v \quad Values(x) = o}{p_x = \psi(|objsWithAttrVal(\text{“}f\text{”}, v)|) \wedge p_{o.f} = HIGH} \quad [AR]$$

$$\frac{s : y = x[i] \ \ Values(x) = o \ \ Values(i) = v_i \ \ Values(y) = v}{\begin{array}{l} p_x = \psi(|cltsWithIdxVal(v_i, v)|) \wedge \\ p_i = \psi(|idxesWithVal(v, o)|) \wedge p_{o[v_i]} = HIGH \end{array}} \quad [SR]$$

$$\frac{s : \texttt{if}\ x == y\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \quad Values(x) = Values(y)}{p_x = HIGH \wedge p_y = HIGH} \quad [EQ]$$

$$\frac{s : \texttt{if}\ x \neq y\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \quad Values(x) \neq Values(y)}{p_x = 0.5 \wedge p_y = 0.5} \quad [NEQ]$$

Function Definitions:

$\psi(n) = 0.5 + 0.5 \times (2 \times HIGH - 1) \times \frac{1}{n}$

$Values(x)$ the value of x observed during execution

$objsWithAttrVal(a, v)$ returns objects having attribute$a$ with value v

$objsWithAttr(a)$ returns the objects having attribute a

$cltsWithIdxVal(i, v)$ returns collections having value v at index i

$idxesWithVal(v, c)$ returns indexes containing value v in collection c

Figure 4: Variable Constraints.

5 CONSTRAINT GENERATION

In this section, we present how to generate probabilistic constraints. As discussed in Section 3, our analysis consists of two phases. In the first phase, we generate variable constraints and in the second, we generate statement instance constraints.

5.1 Variable Constraint Generation

In the first phase, constraints are constructed from the execution trace to model the probabilities of variables and branch predicate states. Note that in this phase we only reason about the correctness/faultiness of variables, which will be used to infer statement instance correctness/faultiness in the next phase. Intuitively, our design can be understood as first finding the correctness/faultiness of variables, then the statement instances that have operands likely correct but the result variable likely faulty are likely buggy.

Encoding Rules. Fig. 4A presents the encoding rules. Rule [1] encodes the forward causality of a statement instance e, denoting the

propagation of the likelihoods (of being correct) from the control dependence predicate *pred* (i.e., e control depends on *pred*), the uses (i.e., rhs operands of e) to the definition (i.e., lhs variable def). Intuitively, if the uses and the control dependence are correct, there is good chance the lhs variable has a correct state. Rule [2] encodes the backward causality, denoting the propagation of correctness likelihood from the lhs variable to *one* of the used variables. Intuitively, if the lhs variable, the control dependence, and all the other rhs operands are correct, the remaining rhs operand u has p_u (not *HIGH*) probability to be correct. Different kinds of statements have different p_u values. Details will be explained later. Rule [3] represents the causality for predicates. If all the related variables in a statement instance e have correct states, the control dependence *pred* is likely correct (i.e., the correct branch has been taken).

Computation Rules of the Backward Propagation Probability p_u. Fig. 4B presents part of the computing rules of p_u (in Rule [2]). The essence of these rules is to determine p_u based on whether the computation is a one-to-one or many-to-one mapping. Rule [1TO1] specifies the computation rule for statements denoting one-to-one mappings. In this case, p_u is *HIGH* to indicate if the lhs is correct, a rhs operand is determined to be correct when all other rhs operands are correct (e.g., when $z = x + y$, z and x likely correct indicates y likely correct). [MOD] specifies for a mod operation $z = x \bmod y$, z and y being correct does not hint x is correct. This is reflected by $p_x = UNCERTAIN$ (0.5), which means there is no evidence indicating the correctness of x. Intuitively, given a specific y value (e.g., $y = 10$), there are many possible values of x that yield the same z value (e.g., $z = 1$) by $z = x \bmod y$ (e.g., x=11, 21, 31, ...). Hence, x's correctness is unknown.

[AR] is for an attribute read. It means that y and x correct indicates the attribute read $o.f$ is likely correct (i.e., $p_{o.f} = HIGH$) with o the object in x. On the other hand, if y and $o.f$ correct, the likelihood of x being correct depends on the number n of objects that have the same f field and the same field value v. It is computed as $\psi(n)$. Particularly, when $n = 1$, the probability is *HIGH*. If n is large, the probability tends to be 0.5 (i.e., correctness is unknown). The rule for attribute write is similar and hence elided.

[SR] specifies the rule for an element read of a collection, which includes three uses, i, x, and $x[i]$ and one definition y. The rule means when y, i, and $x[i]$ are correct, the likelihood of x correct, p_x, depends on the number of collection objects that have value v at index v_i; when y, $x[i]$, and x are correct, the likelihood of i being correct, p_i, depends on the number of indexes in the collection object o that store the same value v; when y, x, and i are correct, the array element is likely correct. We use $p_{o[v_i]}$ instead of $p_{x[i]}$ because during inference, the random variable is associated with the collection element $o[v_i]$ instead of the symbol $x[i]$.

[EQ] is the rule for equivalence check, which has two uses x and y and a definition, namely, the branch outcome. It means that if x and y are equivalent, the branch outcome and x correct indicates y correct (i.e., $p_y = HIGH$), and similarly the branch outcome and y correct indicates x correct. Intuitively, equivalence relation denotes a one-to-one mapping. For instance, assume $x == 10$ yielding true is correct and 10 is correct, a faulty x cannot yield the correct branch outcome. In contrast, [NEQ] indicates that if x and y are inequivalent, the branch outcome and x being correct does not suggest y correct. Intuitively, inequivalence indicates a many-to-one mapping. For instance, assume $x \neq 10$ yielding true is correct and 10 is correct, there are many values of x that can yield the correct branch outcome and hence we cannot gain any confidence about the correctness of x. The rules for other comparative operations (e.g., > and <) are similarly defined.

A. Variable – to – Statement Constraints

$$\mathcal{P}_e(pred) \wedge \mathcal{P}_e(def) \wedge \bigwedge_{u \in Uses(e)} \mathcal{P}_e(u) \xrightarrow{HIGH} \mathcal{P}(inst_e) \quad [1]$$

$$\mathcal{P}_e(pred) \wedge \neg\mathcal{P}_e(def) \wedge \bigvee_{u \in Uses(e)} \neg\mathcal{P}_e(u) \xrightarrow{HIGH} \mathcal{P}(inst_e) \quad [2]$$

$$\mathcal{P}_e(pred) \wedge \neg\mathcal{P}_e(def) \wedge \bigwedge_{u \in Uses(e)} \mathcal{P}_e(u) \xrightarrow{HIGH} \neg\mathcal{P}(inst_e) \quad [3]$$

B. Program Structure Constraints

$$\mathcal{S}(inst_e) \xleftrightarrow{HIGH} \mathcal{P}(inst_e) \qquad \mathcal{S}(inst_e) = 1\ (sp_e) \quad [4]$$

$$sp_e = \frac{1}{2} \times [\underbrace{\phi_1(\frac{|sliceInFunc(e)|}{|fullSlice()|})}_{①} + \underbrace{\phi_2(\frac{|instsNotInSlice(e)|}{|allInsts(e)|})}_{②}]$$

C. Naming Convention Constraints

$$\mathcal{N}(inst_e) \xleftrightarrow{HIGH} \mathcal{P}(inst_e) \qquad \mathcal{N}(inst_e) = 1\ (np_e) \quad [5]$$

$$np_e = \frac{1}{2} \times [\underbrace{\phi_1(simFunc(e))}_{①} + \underbrace{\phi_1(simStmt(e))}_{②}]$$

Function Definitions:
$\phi_1(x) = 0.5 - 0.5 \times (2 \times HIGH - 1) \times x$
$\phi_2(y) = 0.5 + 0.5 \times (2 \times HIGH - 1) \times y$
$sliceInFunc(e)$ returns the instances in slice that belong to the function of e.
$fullSlice()$ returns all the instances in the slice.
$instsNotInSlice(e)$ returns instances of the statement of e that are not in the slice.
$allInsts(e)$ returns all the instances of the statement of e.
$simFunc(e)$ calculates the similarity between the function name of e's statement and the faulty output variable's name.
$simStmt(e)$ returns the average similarity between each involved variable name in e with the faulty output variable's name.

Figure 5: Statement Instance Constraints.

5.2 Statement Instance Constraint Generation

We generate three kinds of constraints for statement instances: (1) *variable-to-statement constraints* modeling the causality between variable probabilities and statement instance probabilities; (2) *program structure constraints* modeling hints from program structure; (3) *naming convention constraints* modeling hints from names. We reason about statement instances instead of statements because an instance of a faulty statement may not induce any faulty state at all, for example, '$x > 10$' being mistaken as '$x >= 10$' does not induce any faulty states for any $x \neq 10$. In this case, we consider the instance as correct. Hence in this paper, we do not allow the likelihood of an instance being correct/faulty to directly affect the likelihood of another instance of the same statement.

Variable-to-Statement Constraints. Constraint [1] in Fig. 5A denotes that if all the variables involved in a statement instance e, including the control dependence, the definition and all the uses are correct, the instance is likely correct. Constraint [2] represents that if the control dependence is correct, but the definition and at least one use are faulty, the instance is likely correct. This is because the root cause must happen before the current instance. Constraint [3] denotes that when all uses are correct but the definition is faulty, the statement instance is likely faulty (i.e., the root cause). We use the posterior variable probabilities from the previous phase as the prior probabilities of variables in this phase.

Program Structure Constraints. From program structure, we can extract hints to enhance inference. Given an instance e of statement s, we consider the following two kinds of hints from program structure. First, at the function level, we focus on how

many instances in the slice belong to the function of s. The larger the number, the more suspicious e is, because the faulty output is more likely composed in the function. Second, at the statement level, we consider how many other executed instances of statement s are not in the slice. Specifically, if s is executed many times but only a few of them are included in the slice of the failure, the statement instance is less likely faulty. For example, some low-level utility functions (e.g., `make_str()` in the Fig. 1) are called many times at different points and only a few of them are related to the failure. Statement instances in these low level functions are unlikely suspicious.

Fig. 5B presents the program structure constraints where $\mathcal{S}(inst_e)$ asserts that instance e is correct from the program structure perspective. The prior probability sp_e of $\mathcal{S}(inst_e)$ is computed by the average of two parts. Part ① computes the probability at the function level. If many instances in the function of e are included in the slice, the probability tends to be LOW, denoting they are suspicious. Otherwise, it tends to be 0.5. Part ② computes the probability at the statement level. If many instances of the statement of e are not in the slice, the probability tends to be $HIGH$, suggesting correctness. Otherwise, it tends to be 0.5.

Naming Convention Constraints. Given a faulty variable x and an instance e of statement s, we collect two kinds of naming convention hints. First, we measure how similar the function name of statement s is to the name of faulty output variable x. The higher the similarity, the more suspicious e is. Second, at the statement level, we consider the similarity between the name of the faulty output variable x with variable names in the statement s. When the average similarity is high, the suspiciousness of e is high. For example, in the example of Fig. 1, instances of line 16 are considered less suspicious than instances of line 11 because `"url"` is more similar to { `"purl_str"`, `"purl"`} than to {`"s"`, `"i"`}.

Fig. 5C presents the naming convention constraints where $\mathcal{N}(inst_e)$ asserts that instance e is correct from the naming convention perspective. The computation of the prior probability np_e of $\mathcal{N}(inst_e)$ consists of two parts. Part ① computes the probability from the function level naming convention. Intuitively, if the similarity is high, the probability approaches $1 - HIGH$. Otherwise, the probability approaches 0.5. Part ② computes the probability from the statement level naming convention.

The computation of lexical similarities between strings is similar to the one illustrated in Section 5.4 in our prior work [51]. We omit the details due to the space limitation.

6 EVALUATION

We implement a prototype in Python for debugging Python programs. We choose Python as it has become one of the most popular programming languages [7]. We aim to address the following research questions in the evaluation.

RQ1: How effective and efficient is our approach in assisting debugging real-world bugs?

RQ2: How does our tool compare to a recent interactive debugging technique [24] that does not encode human intelligence?

RQ3: What is the impact of the threshold $HIGH$, which is the only tunable parameter in our tool.

RQ4: What is the impact of the different kinds of statement instance constraints?

RQ5: How can our tool improve human productivity in practice?

To answer RQ1-RQ4, we apply our technique to a set of real-world bugs, as shown in Table 3 (columns 1-4). Our benchmarks consist of two kinds of bugs. The first kind (I) includes 25 bugs from 10 popular Python projects on GitHub and the second kind (II) includes 10 algorithmic bugs posted on Stack Overflow. As shown in column 3 of Table 3, some of the projects are among the largest Python projects one can find on GitHub, with the largest having over 54K LOC. The collected project bugs (I) also denote the typical complexity level. We use the failure inducing inputs in the bug reports. When collecting these bugs, we considered the diversity of projects. They mainly fall into the following categories.

`Fabric` – remote deployment and administration application.

`Requests` and `urllib3` - very popular HTTP libraries.

`Simplejson` - a widely used JSON encoding/decoding library.

`Bottle`, `flask` and `web2py` - widely used web application system development frameworks.

`Werkzeug` - a popular WSGI utility library for Python.

`Dateutil` - an extension to the `datetime` module of Python.

`Scrapy` - a well known web crawler.

For algorithmic bugs (II), the programs mainly implement various algorithms from Project Euler [5] and LeetCode [6]. Their issue numbers on `stackoverflow.com` are shown in column 4.

Table 3 shows the summary of our experimental results. Columns 5-6 present the numbers of events and sliced events. Observe that many slices are so large that manual inspection would be difficult. The events for project bugs are smaller than we expected because the failure inducing inputs in bug reports tend to be small as they have gone through input reduction (e.g., using delta-debugging [8]). This is very typical for Python bugs reported on GitHub. Columns 7-9 report the number of solved constraints by our tool. Columns 7 (`Variable`) and 8 (`StmtInst`) represent the numbers of solved variable constraints and statement instance constraints, respectively. Column 9 reports their total number.

Effectiveness and Efficiency (RQ1). To evaluate the effectiveness of our tool, we set the threshold $HIGH = 0.95$ and debug the benchmark programs following the recommendations suggested by our tool. We count the interactions with the tool (i.e., the recommended instances that we have to inspect and provide feedback) till the root causes are located. Column 13 (`PD`) reports the number of interactions, including the final root cause step. The results show that our approach is highly effective. We only need to inspect a small number of steps (on average 3 steps for project bugs and 5 for algorithmic bugs). For a few cases (e.g., fabric#166 and requests#1462), our tool can directly report the root cause. Note that this does not suggest the algorithmic bugs are more complex. In fact, they are mostly loop intensive and hence their failure propagation paths often involve many iterative steps. As a result, our tool may require the user to provide feedback for multiple instances of the same statement in different iterations.

Columns 10-12 present the tracing time, solving time and total time, respectively. The solving time is the sum of all the inspection rounds. The results show that our approach is highly efficient. Especially, our tool can complete inferences in a second for most cases. Note that tracing time is a one time cost for each failure.

Comparison with *Microbat* [24] (RQ2). Since [24] does not support Python, we re-implemented it in Python for the comparison.

Table 3: Summary of the Experiment Results

Benchmarks				Trace		Solved Constraints			Time (sec)			Inspected Steps	
	Projects/Algs.	SLOC	Issue#	Events	Sliced Events	Variable	StmtInst	All	Tracing	Solving	Total	PD	FD
I	Fabric	3061	882	6219	88	599	189	788	3.36	0.1	3.46	2	4
			166	120	42	153	56	209	0.98	0.03	1.01	1	6
			610	345	39	254	126	380	1.12	0.09	1.21	2	5
			898	6199	80	271	126	397	3.34	0.06	3.4	1	10
	Requests	10595	1462	4549	128	266	70	336	2.67	0.04	2.71	1	6
			1711	4922	12	167	42	209	2.83	0.1	2.93	2	1
			2613	8084	886	1009	406	1415	10.85	0.25	11.1	2	17
			2638	7509	876	363	147	510	11.03	0.21	11.24	5	14
			395	1912	49	509	182	691	1.26	0.23	1.49	2	6
			1767	2047	652	7975	2303	10278	0.76	0.22	0.98	5	21
			2638	7509	876	356	126	482	28.91	0.14	29.05	5	15
			3017	895	32	220	56	276	0.76	0.12	0.88	3	2
	simplejson	4104	81	434	164	633	189	822	0.16	0.17	0.33	3	5
			85	683	174	1472	511	1983	0.25	0.34	0.59	5	10
	Flask	2658	857	1684	23	36	21	57	0.81	0.03	0.84	1	2
	bottle	2569	595	1138	481	3048	2457	5505	0.65	0.83	1.48	3	20
			633	708	321	1020	742	1762	0.64	0.21	0.85	2	21
	werkzeug	11009	99	2605	50	269	147	416	1.43	0.12	1.55	4	12
	dateutil	3647	234	2093	152	240	70	310	0.95	0.12	1.07	4	8
			229	1306	144	293	105	398	0.44	0.13	0.57	4	10
	scrapy	11145	861	4211	861	7452	2128	9580	8.61	1.58	10.19	8	12
			24	3970	1008	8190	3136	11326	9.61	2.14	11.75	5	14
	web2py	54479	742	4030	34	102	56	158	17.77	0.08	17.85	3	7
			1570	5531	94	145	77	222	21.89	0.04	21.93	2	8
	urllib3	11659	143	997	126	1403	168	1571	4.63	0.06	4.69	4	9
	Average	11493	-	3188	296	1440	533	1973	5.43	0.37	5.80	3	10
II	lcsubstr	19	41623560	400	236	255	77	332	0.08	0.07	0.15	2	5
	happynum	17	30247583	112	68	198	105	303	0.04	0.16	0.2	5	7
	quadratic	26	17681021	249968	2237	592	574	1166	43.14	0.19	43.33	4	20
	primesum	17	17681021	8007	6883	1290	623	1913	1.75	0.4	2.15	3	40
	triangular	21	9182462	7949	4540	2172	1687	3859	1.4	0.6	2	7	X
	fibonaci	18	18794190	284	151	2521	2926	5447	0.07	0.88	0.95	11	X
	lcollatzseq	15	20597369	56625	54192	2711	3080	5791	9.34	0.95	10.29	3	2
	amicable	27	19318000	49780	4265	1241	245	1486	11.22	0.24	11.46	6	6
	mergesort	30	18808105	1046	286	557	91	648	0.24	0.3	0.54	5	5
	euclid	10	16567505	26	9	107	84	191	0.01	0.12	0.13	3	X
	Average	20	-	37420	7287	1164	949	2114	6.73	0.39	7.12	5	-

We call the Python version of *Microbat* the *PyMicrobat*. We compare the performance of the two by counting the number of inspected steps before locating the root cause. To suppress noise (i.e., differences in human feedback), four of the authors debugged each case using both tools independently. We then take the average of the interaction numbers. To achieve fair comparison, our tool requests the same kind of feedback from the developer as *Microbat*, which is to determine whether operands in a statement instance are correct. Column 14 (FD) in Table 3 presents the results of *PyMicrobat*. The results show that for most cases, the number of interactions needed by our tool is much smaller than that by *PyMicrobat*. Observe that *PyMicrobat* failed to locate three algorithmic bugs. The reason is that *PyMicrobat* is strictly based on dynamic slicing, which may miss root causes due to execution omission [31]. In contrast, our tool can fuse additional debugging hints from multiple sources. In requests#1711 and lcollatzseq, our tool requires one more step. Further inspection shows that the two bugs have simple causality so that uncertainty reasoning is not needed.

Impact of Threshold *HIGH* (RQ3). We study the impact of the threshold *HIGH* using three settings, namely, 0.85, 0.90 and 0.95. Fig. 6 presents the variation of the needed steps. Observe that the impact of *HIGH* is small in most cases.

Impact of Constraints (RQ4). We only choose bugs of kind (I) as our subjects as kind (II) bugs mostly do not have function structures and their variable names are almost meaningless. We evaluate the impact of each kind of statement instance constraints by three additional settings, namely, only variable-to-statement instance constraints, variable-to-statement instance constraints with program structure constraints, and variable-to-statement instance constraints with naming convention constraints. Note that variable-to-statement instance constraints are the basis in our model. Fig. 7 shows the variation of the needed steps. Observe that each kind has positive contribution for most cases. The impacts of program structure constraints and naming convention constraints on some cases (e.g., fabric#610 and scrapy#861) are prominent.

User Study (RQ5). We conducted a user study. We selected four well-known algorithmic problems from our benchmark and asked the participants to debug the buggy programs. Table 4 presents the descriptions of the selected bugs. We invited 16 graduate students from the authors' institute to participate the study. To avoid bias caused by programming experience variation, for each program, we randomly partition the students to two groups, one using our tool and the other using the standard Python debugger *pdb* [3].

Table 5 presents the (human) debugging time comparison. The shaded columns present the time of using our tool. Observe that our tool achieves 34.03% speed up on average. To validate the performance difference of the two groups, we introduce the null and alternative hypotheses as follows.

H0: There is no significant performance difference between the two groups.

H1: The performance difference between the two is significant.

We use the Wilcoxon signed rank test to evaluate the null hypotheses and if the p-value is less than 0.05, we will reject it. The last row in Table 5 reports the p-value of each task. Observe that all of them are less than 0.05, which means the performance improvement is significant.

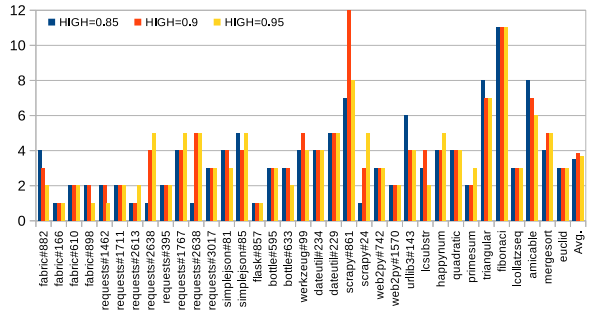

Figure 6: Impact of the Threshold *HIGH*.

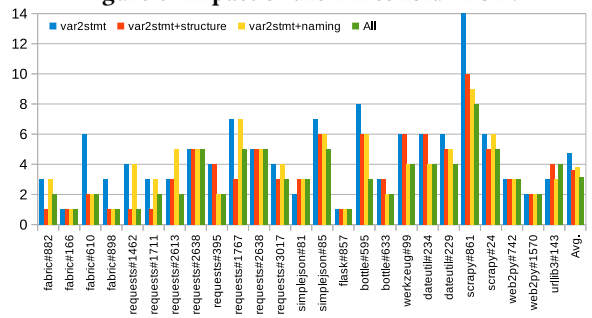

Figure 7: Impact of Each Kind of Constraints.

7 RELATED WORK

Automated debugging and fault localization techniques have been extensively studied. Delta debugging [8–14] leverages a passing execution closely resembling the failing execution as the reference, and uses a sophisticated search algorithm to identify the minimal state differences that trigger the failure. Despite its success, it may be difficult to find a reference execution of high quality in practice. Spectrum based fault localization [15–21] compares program spectra (e.g., code coverage) between the failing and passing executions to identify the most likely root cause. It is highly effective when many runs are available. Statistical fault localization [22, 23] isolates bugs by contrasting the instrumented predicates at particular points. There are also various proposals on improving fault localization and applying fault localization in different areas [33–44, 49]. Our approach is different. First, we assume that only a failing run is available for debugging. Second, these techniques are mostly deterministic and do not handle uncertainty like we do. We shift a lot of decision makings from humans to machines.

Debugging techniques using a single run [24–29, 32] have been proposed as well. For example, Lin et at. [24] developed a feedback-based technique that prunes the search space. But the technique does not reason about uncertainty and hence cannot make use of uncertain hints, which are very useful according to our experiment. Zhang et al. [29] prune dynamic slice with confidence. However, their method of computing confidence is ad-hoc and they do not support probabilistic inference. Many other techniques leverage execution trace for debugging [26–28]. However, most of them do not support human-like reasoning.

There are also techniques leveraging machine learning to locate bugs [45–47]. Dietz et al. [45] trains a graphical model called the Bernoulli graph model using passing traces to determine the most likely faulty code position. Baah et al. [46] proposed probabilistic dependence graph which estimates variable states and learns conditional dependencies from both passing and failing runs to facilitate fault localization. Deng et al. [47] encodes the frequency of execution in a hybrid graphical model named weighted system dependency graph to prioritize heavily trafficked flows to facilitate fault localization. Our technique is different. First, they require a number of failing/passing runs to build the model while our technique only needs a single failing run. Second, these techniques *learn* probabilities from multiple executions whereas our technique *infer* posterior probabilities from prior distributions. Third, their model is static, denoting a program, whereas our model is dynamic, denoting an execution. Fourth, our technique can leverage uncertain hints such as those from structure and names.

Table 4: Debugging Tasks for User Study.

Algorithm	Description	Bug Causality
lcsubstr	Identify the longest increasing substring of a given string.	Indexes incorrectly computed.
quadratic	Find the maximum number of primes satisfying a quadratic formula (Euler problem 27).	Some intermediate results are not updated in iterations.
fibonaci	Find the sum of the even terms of the fibonaci series up to a certain number.	Loop condition is incorrect.
mergesort	Implement the merge sort for a given number sequence.	The comparison operator is wrong.

Table 5: Results of the User Study (Min).

Person\Task	lcsubstr		quadratic		fibonaci		mergesort	
N1/Y1	11.56	6.37	33.1	16.32	15.03	14.55	10.54	12.5
N2/Y2	30.06	8.2	10.14	11.22	27	16.11	18.13	12.23
N3/Y3	10.3	5.29	21.5	10.58	13.31	11.81	15.04	14.32
N4/Y4	5.8	5.5	19.31	17.76	17.24	18.33	18.33	13.9
N5/Y5	2.85	6.96	10.45	14.4	12.45	14.78	20.78	15.61
N6/Y6	21.11	4.89	12.8	10.33	25.69	15.6	20.5	13.8
N7/Y7	23.11	7.36	27.45	14	34.46	13.28	30.5	11.44
N8/Y8	12.71	8.11	20.45	15.4	18.51	18.44	14.33	14.49
Avgerage	14.69	6.59	19.40	13.75	20.46	15.36	18.52	13.54
	55.17%		29.12%		24.92%		26.91%	
Overall	34.03%							
P-Value	0.006		0.037		0.049		0.014	

8 CONCLUSION

We propose a probabilistic inference based debugging technique. We model the debugging problem as two phases of probabilistic inference: (1) inferring variable correctness/faultiness probabilities and (2) inferring statement instance correctness/faultiness probabilities. Our technique allows us to debug by leveraging various hints from execution trace, program structure, variable names and human feedback. The results show that our technique can identify root causes of a set of real-world bugs in a few steps, much faster than a recent proposal that does not encode human intelligence. It also substantially improves human productivity.

ACKNOWLEDGMENTS

This research was supported by DARPA under contract FA8650-15-C-7562, NSF under awards 1748764, 1409668 and 1320444, ONR under contracts N000141410468 and N000141712947, Sandia National Lab under award 1701331, National Basic Research Program of China No.2014CB340702, National Natural Science Foundation of China No.61472175, 61472178 and 61772263, and Natural Science Foundation of Jiangsu Province of China BK20140611. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] libdai. https://staff.fnwi.uva.nl/j.m.mooij/libDAI/
[2] requests. http://docs.python-requests.org/en/master/
[3] pdb. https://docs.python.org/2/library/pdb.html
[4] Pycharm. https://www.jetbrains.com/pycharm/
[5] Project Euler. https://projecteuler.net/
[6] Leetcode. https://leetcode.com/
[7] IEEE Spectrum http://spectrum.ieee.org/computing/software/the-2017-top-programming-languages
[8] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 253–267, 1999.
[9] A. Zeller. Isolating cause-effect chains from computer programs In *Proceedings of the 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 1-10, 2002.
[10] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. In *IEEE Transaction on Software Engineering (TSE)*, 28(2):183–200, 2002.
[11] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 342–351, 2005.
[12] G. Misherghi and Z. Su. Hdd: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 142–151, 2006.
[13] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. Darwin: An approach for debugging evolving programs. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE)*, pages 33–42, 2009.
[14] W. N. Sumner, Y. Zheng, D. Weeratunge and X. Zhang. Precise calling context encoding. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 525-534, 2010.
[15] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 88–99, 2009.
[16] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of International Conference on Automated Software Engineering (ASE)*, pages 30–39, 2003.
[17] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 432–449, 1997.
[18] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 45–55, 2009.
[19] L. Gong, D. Lo, L. Jiang, and H. Zhang. Interactive fault localization leveraging simple user feedback. In *IEEE International Conference on Software Maintenance (ICSM)*, pages 67–76, 2012.
[20] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. van Gemund. A practical evaluation of spectrum-based fault localization. In *Journal of Systems and Software (JSS)*, 82(11):1780 – 1792, 2009.
[21] D. Gopinath, R. N. Zaeem and S. Khurshid. Improving the effectiveness of spectra-based fault localization using specifications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. pages 40-49, 2012.
[22] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 15-26, 2005.
[23] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical Debugging: A Hypothesis Testing-based Approach. In *IEEE Transactions on Software Engineering*, 32(10):831-848, 2006.
[24] Y. Lin, J. Sun, Y. Xue, Y. Liu and J. Dong. Feedback-Based Debugging. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, (to appear), 2017.
[25] A. J. Ko and B. A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 151–158, 2004.
[26] A. J. Ko and B. A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 301–310, 2008.
[27] G. Pothier and . Tanter. Back to the future: Omniscient debugging. *IEEE Software*, 26(6):78–85, 2009.
[28] J. Ressia, A. Bergel, and O. Nierstrasz. Object-centric debugging. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 485–495, 2012.
[29] X. Zhang, N.Gupta and R. Gupta. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 169-180, 2006.
[30] J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations, *Exploring articial intelligence in the new millennium 8*, 2003.
[31] X. Zhang, S. Tallam, N. Gupta and R. Cupta Towards locating execution omission errors. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 415-424, 2007.
[32] Tao Wang and Abhik Roychoudhury Hierarchical dynamic slicing In *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA)*, pages 228-238, 2007
[33] B. Baudry, F. Fleurey and Y. L. Traon. Improving test suites for efficient fault localization In *Proceedings of the 28th international conference on Software engineering (ICSE)*, pages 82-91, 2006
[34] B. Liu, Lucia, S. Nejati, L. C. Briand and T. Bruckmann. Simulink fault localization: an iterative statistical debugging approach. *Software Testing, Verification and Reliability*, 26(6): 431-459, 2016.
[35] D. Hao, T. Xie, L. Zhang, X. Wang, J. Sun and H. Mei. Test input reduction for result inspection to facilitate fault localization. *Automated software engineering*, 17(1): 5-31, 2010.
[36] X. Li, M. Amorim and A. Orso. Iterative User-Driven Fault Localization. In *Haifa Verification Conference*, pages 82-98, 2016.
[37] W. Jin and A. Orso. F^3: Fault localization for Field Failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, pages 213-223, 2013.
[38] L. Zhang, L. Zhang and S. Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA)*, pages 765-784, 2013.
[39] F. S. Ocariza, G. Li, K. Pattabiraman and A. Mesbah. Automatic fault localization for client-side JavaScript. In *Software Testing, Verification and Reliability*, 26(1): 69-88, 2016.
[40] S. Wang, D. Lo, L. Jiang, Lucia and H. C. Lau: Search-based fault localization. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 556-559, 2011.
[41] G. Birch, B. Fischer and M. Poppleton. Using Fast Model-Based Fault Localisation to Aid Students in Self-Guided Program Repair and to Improve Assessment. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, pages 168-173, 2016.
[42] X. Xia, L. Gong, T. B. Le, D. Lo, L. Jiang and H. Zhang. Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. In *Automated Software Engineering (ASE)*, 23(1): 43-75, 2016.
[43] C. Zhang, D. Yan, J. Zhao, Y. Chen and S. Yang. BPGen: An Automated Breakpoint Generator for Debugging. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 271-274, 2010.
[44] J. Al-Kofahi, H. V. Nguyen and T. N. Nguyen. Fault Localization for Make-Based Build Crashes. In *roceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 526-530, 2014
[45] L. Dietz, V. Dallmeier, A. Zeller, and T. Scheer. Localizing bugs in program executions with graphical models. In *Neural Information Processing Systems (NIPS)*, 2009.
[46] G. K. Baah, , A. Podgurski and M. J., Harrold. The probabilistic program dependence graph and its application to fault diagnosis. *IEEE Transactions on Software Engineering (TSE*, 36(4): 528-545, 2010.
[47] F. Deng and J.A. Jones. Weighted system dependence graph. In *Proceedings of the Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 380–389, 2012.
[48] S. Zhang and C. Zhang. Software bug localization with markov logic. In *Proceedings of the 36th International Conference on Software Engineering (ICSE14)*, pages 424–427, 2014.
[49] T. B. Le, D. Lo, C. L. Goues and L. Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, pages 177-188, 2016.
[50] N. E. Beckman and A. V. Nori. Probabilistic, modular and scalable inference of typestate specifications. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages:211-221, 2011.
[51] Z. Xu, X. Zhang, L. Chen, K. Pei and B. Xu Python Probabilistic type inference with natural language support. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages:607-618, 2016.