Simple Fault Localization using Execution Traces

Anonymous Authors

***Abstract*—Traditional spectrum-based fault localization (SBFL) exploits differences in a program's coverage spectrum when run on passing and failing test cases. However, such runs can provide a wealth of additional information beyond mere coverage. Working with thousands of execution traces of short programs submitted to competitive programming contests and leveraging machine learning and additional runtime, control-flow and lexical features, we present simple ways to improve SBFL. We also propose a simple trick to integrate context information. Our approach outperforms SBFL formulae such as Ochiai on our evaluation set as well as QuixBugs and requires neither a GPU nor any form of advanced program analysis. Existing SBFL solutions could possibly be improved with reasonable effort by adopting some of the proposed ideas.**

***Index Terms*—fault localization, automated program repair**

I. Introduction

The goal of fault localization (FL) is to spot bugs in software. This can save development cost and time as well as ease software maintenance. Fault localization is also an important step in automated program repair where repair effort is concentrated on code locations found suspicious (i.e., possibly buggy) by a fault localization system.

A commonly employed fault localization method is spectrum-based fault localization (SBFL). While many different kinds of program spectra exist [1], most of the time, SBFL relies on the coverage or hit spectrum of the target program. This spectrum indicates which program elements are covered, that is executed, by passing or failing test cases. Program elements can comprise statements, lines, blocks or even methods. However, in this work we focus on the line level. From the hit spectrum, four central features are derived *for each line* l (or other program element) of a program:

e_p, the number of passing test cases covering line l,
e_f, the number of failing test cases covering line l,
n_p, the number of passing test cases *not* covering line l,
n_f, the number of failing test cases *not* covering line l.

By weighing these four features by means of a risk evaluation formula [2] (also referred to as ranking metric) such as Ochiai [3], [4] each line is assigned a score. Lines can then be ranked by this score (usually high to low) to find the most likely fault locations.

This work: In this work we combine three simple ideas to improve spectrum-based localization performance. First, we include features from *additional program spectra* (e.g., the count spectrum) that can easily be obtained from execution traces and that do not require any form of advanced static or runtime analysis. Second, we take *context* into account. That is, the calculation of the score for a line l may consider features belonging to surrounding lines (within a sliding window). Finally, instead of simple formulae we follow a *data-driven* approach and use extracted features to fit a gradient boosted machine (GBM). Section II contrasts these points with previous work.

We rely on simple Python programs from the RunBugRun [5] dataset (see Section III for details). We also evaluate our model on the Python version of the QuixBugs dataset [6] and provide a simple ablation study (Section IV-A).

Results: On both, our evaluation set of RunBugRun bugs as well as on QuixBugs [6], our GBM model outperforms commonly used formulae taken from well-known SBFL systems such as Ochiai [4], DStar [7] and Tarantula [8]. Results are detailed in Section IV.

Conclusions: Our work suggests that SBFL performance can be improved by relatively simple means; in particular, by using a count spectrum instead of a hit spectrum and a rolling window over multiple lines to capture some form of context. These ideas are relatively simple to implement and could be integrated into existing SBFL systems and frameworks. In contrast to other recent deep learning-based approaches, the used GBM models are quick to train, have modest hardware requirements and do not require a GPU.

II. Related Work

We focus on previous work related to the most important aspects of our approach, that is i) previous learning-based FL and in particular SBFL methods, ii) previous methods exploiting alternative or multiple spectra, iii) approaches that address the problem of missing context in SBFL and finally iv) methods that rely on traces.

Learning-based: Certainly, the idea of fitting a SBFL score function is not new. Xuan and Monperrus [9] propose MULTRIC, that combines multiple ranking formulae using learned weights. Yoo [10] uses genetic programming to evolve an optimal risk evaluation formula. Early experiments with neural networks were done by Wong and Qi [11]. In recent years, deep learning has been successfully applied to the problem of fault localization in various different forms [12], [13], including large code language models [14], [15] and graph neural networks (GNNs) [16]–[20]. Our work employs GBMs to predict fault scores on a per-line level.

Additional spectra & features: Yilmaz, Paradkar, and Williams [21] use the time spectrum (i.e., execution time features) for localization. This work also employs some form of learning, as it uses Gaussian Mixture Models (GMMs) to model time. Zhang, Chan, Tse, *et al.* [22] rely on a control flow spectrum (edge spectrum) to localize faults. They focus on cases where the effect of a fault may show at a different location than its true cause. Santelices, Jones, Yu, *et al.*

[23] devise a localization approach that uses def-use pairs and a branch coverage spectrum. DEPUTO [24] improves spectrum-based fault localization through the use of abstract interpretation. FLUCCS [25] uses code change metrics such as age, churn or complexity to aid localization. DeepFL [12] combines features from various different localization methods, including SBFL, mutation-based fault localization (MBFL) as well as code complexity and text features. In this work, in addition to coverage and count spectrum features, we also use very simple control flow features. We also experiment with lexical features.

Using context: Traditional SBFL methods determine suspiciousness separately for each program element without taking into account possible dependencies between program elements or other contextual information. This problem has been identified in previous work [26] and several attempts have been made to tackle it. We should note that context is taken here in a very broad sense and includes several different ways to factor dependencies between program elements or simply multiple program elements into the suspiciousness score calculuation. Zhao, Wang, and Yin [27] propose calculating suspiciousness scores for control flow edges. We also experiment with simple control flow features, however, without carrying out any advanced control flow analysis. Like our work, Laghari and Demeyer [28] use a sliding a window, albeit to mine call sequences from call traces. We use sliding windows and simple control flow features to "contextualize" per-line features.

While DeepFL [12] employs LSTMs, the spacial dimension of the LSTM (often referred to as time dimension) seems not to be used to capture context over multiple lines or statements but to combine features of different types belonging to a single program element. In contrast, Li, Wang, and Nguyen [13] convert the spectrum information into images (or rather a feature map) and use them to train convolutional neural networks (CNNs); here one spatial dimension seems to span multiple program elements. Similarly, Zhang, Lei, Mao, *et al.* [29] train a series of different deep learning architectures, including CNNs, LSTMs and MLPs, on spectral data where one of the input dimensions can span multiple program statements or lines, thus allowing the network to model some form of basic context. CAN Zhang, Lei, Mao, *et al.* [18] aim to contextualize FL by modeling dependencies between program elements using graphs and GNNs. Finally, some of the previously mentioned work, for instance, the use of edge spectra [22] could also be said to capture some form of context. While we are not using neural networks, our sliding window technique is somewhat reminiscent of convolution, used in CNNs.

Using traces: Not much work explicitly mentions the use of execution traces. However, similar types of information may be used implicitly, e.g., for control flow analysis or even to determine coverage. AMPLE [30] and SPEQTRA [31] both use method call traces to locate faults at the class level. Similarly, He, Ren, Zhao, *et al.* [32] employ tracing to collect not only coverage but also call relation information. SmartFL [33] uses probabilistic modeling on information which is, in addition to static analysis, also obtained from execution traces.

III. Methodology

Our approach involves five main steps: 1) obtaining execution traces, 2) calculating line-level features, 3) calculating feature windows, 4) training and finally 5) evaluation. We detail each step below.

1) Obtaining traces: In our work we target buggy Python programs from the RunBugRun [5] dataset which itself is based on the CodeNet dataset [34] and contains programs originally stemming from programming competitions such as AtCoder. We select from RunBugRun a subset of Python bugs with at least one failing and one passing test case. Each bug of said subset is instrumented and executed on all its test cases (the test cases are also part of RunBugRun). For instrumentation, we rely on Python's `sys.settrace`[1] system. This allows us to install a trace callback that is called whenever the Python interpreter is about to execute a line of program code. We use this callback to collect the following information (for each program step): 1) the number of the line being currently executed, 2) the number of the previously executed line, 3) the current value of "primitive" local variables (`int`, `bool`, `float`), 4) the length of strings, lists, sets and numpy arrays 5) a high resolution timestamp, 6) a step counter (i.e., number of total executed steps or lines so far).

In this way, we obtain traces for over 24,000 bugs. These traces record up to millions of execution steps and their *compressed* file sizes range from a few kilobytes to several hundreds of megabytes in extreme cases. In order to be suitable as input for a machine learning algorithm this data must first be condensed as described below. Importantly, in this work we only use line number information, that is items 1) and 2); all of our features can be derived from these two data points.

2) Calculating line-level features: To condense traces we aggregate trace records by bug ID and test outcome (pass or fail) and calculate the following *per-line* features.

Execution counts and coverage (spectrum) We calculate e_p, e_f as described earlier, as well as N_f and N_p, the total number of passed and failed tests, respectively. Moreover, we also calculate the corresponding count spectrum, that is, not only whether a certain line l was executed but also how often. We also include normalized versions of these features, i.e., the count spectrum divided by the total number of executions. Similarly, p_p and p_f, pass and fail rate, are the normalized versions of e_p and e_f. Note that all of the above features can be derived from the line number information. For instance, if a specific line l appears in the trace it was hit and is thus covered. Similarly, by counting occurrences of l we can obtain a count spectrum.

SBFL formulae (spectrum) We calculate ranking scores using Ochiai [4], DStar[2] [7] and Tarantula [8]

[1]https://docs.python.org/3/library/sys.html#sys.settrace

as follows and use each one as a feature:

$$Ochiai = \frac{e_f}{\sqrt{N_f \cdot (e_f + e_p)}}$$

$$DStar^2 = \frac{e_f^2}{e_p + (N_f - e_f)}$$

$$Tarantula = 1 - \frac{h}{h + e_f/N_f} \; where \; h = \frac{e_p}{N_p}$$

For all divisions, we add a small ϵ to the denominator to avoid division by zero.

Incoming/outgoing paths (control flow) We use the line number to calculate some *very simple* control flow features. Remember that our traces contain pairs (l, l') of the current and previously executed line number. We calculate for each line l the minimum, maximum, mean and median difference $l - l'$. If control flow is linear (i.e., line l is always executed right after l') these features will all evaluate to 1; however, if non-linear control flow occurs at these lines, the difference between l and l' will be larger than 1 (if we have a forward jump from l' to l) or even negative (for backward jumps). Next, we calculate the number of outgoing and incoming "paths" by counting for a line l the number of different l' and vice versa. These two features represent the control flow "fan-in" and "fan-out" for a line; put simply, they approximate how non-linear control flow is at a particular line. Of course, we calculate two different versions for each of these features, one for passing and one for failing tests.

Control flow keywords (lexical) Finally, we also include simple lexical features. For each line l we match a simple regular expression against the corresponding code line to determine if it contains control flow-related Python keywords such as `if`, `else`, `for`, `while` and so on. This results in a single boolean feature for each keyword.

3) Calculating feature windows: So far, we have calculated features on a per-line basis. That is, for each line with line number l in a program we have a feature vector $\mathbf{x_l} = (x_1^l, x_2^l, \ldots, x_n^l)$. We could fit a statistical model to predict a suspiciousness score for each such vector. However, we conjecture that neighboring lines affect the suspiciousness score of a line. Consequently, we combine *neighboring* feature vectors into a single feature vector. This is done by sliding a window of size w over each line vector (in ascending order of line numbers). This way, for a line l, which we call the *focal line*, we obtain a *contextualized* feature vector

$$\begin{aligned}\mathbf{x_l^+} = (&x_1^{l-\lfloor w/2 \rfloor}, \ldots, x_n^{l-\lfloor w/2 \rfloor},\\ &\cdots,\\ &x_1^l, \ldots, x_n^l,\\ &\cdots,\\ &x_1^{l+\lfloor w/2 \rfloor}, \ldots, x_n^{l+\lfloor w/2 \rfloor})\end{aligned}$$

where x_i^l denotes the i^{th} feature of line l. We introduce padding lines at the beginning and at the end such that neighboring line vectors are also defined for the first and last lines. The idea of combining neighboring features is somewhat reminiscent of the concept of local context in neural program repair [35]. For each spectral feature, we also include the maximum and minimum value within the window as additional feature.

4) Training: We frame the localization problem as a binary classification task. Buggy lines l will be assigned the positive class ($y_l = 1$), all other lines the negative class ($y_l = 0$). We assign classes based the ground truth patch of insertions and deletions (part of RunBugRun). In particular, for insertions of new lines we use a heuristic that defines the closest previous line in the buggy program as buggy. Of our roughly 24,000 bugs we use 90 % for training and the remaining 10 % for validation (i.e., as validation set). Note that we call this validation set instead of test set as we do "introspective" analysis (e.g., feature importance) on this data, which, strictly speaking, one is not supposed to do on actual test sets. Then, we train a gradient boosting machine on pairs of contextualized vectors and class labels $(\mathbf{x_l^+}, y_l)$. For this, we use the LightGBM[2] framework which, in early experiments, performed slightly better than CatBoost[3]. Likewise, training under the LambdaRank [36] objective performed worse than our windowed vectors and was thus not pursued further.

The class distribution is heavily skewed towards the negative class (roughly by a factor of 10). We try resampling using SOMTE-ENN [37] as implemented by the Imbalanced-learn framework [38] without much success. Resampling has, however, been successfully applied in FL in previous work [29].

5) Evaluation: We use the probability assigned to the positive (i.e., buggy) class as a score for suspiciousness. For each bug we rank all lines in descending order of suspiciousness. Ties are broken by assigning the lower rank to the first line in line number order, this way no rank is shared by multiple lines. In line with previous work [12], [13], [18]–[20] we use the following three evaluation measures where $\hat{r}_l$ is the predicted rank for line l and $\hat{r}^*$ is the lowest rank $\hat{r}_l$ with $y_l = 1$, that is the lowest rank assigned to any buggy line for a specific bug. Note that the lowest possible rank is one.

Mean First Rank (MFR) We calculate MFR as the mean of all $\hat{r}^*$ over all bugs under evaluation.

Mean Average Rank (MAR) The calculation of the MAR measure is similar: for each bug under evaluation we calculate the average of *all* ranks $\hat{r}_l$ with $y_l = 1$. Then we calculate the mean of such averages over all bugs under evaluation.

Top-N Top-N is calculated as the portion of bugs under evaluation where $\hat{r}^* \leq N$. We calculate this measure for $N = 1$, 3 and 5.

Baselines: We use previously defined Ochiai, DStar[2] and Tarantula ranking metrics as baselines. An additional random baseline assigns suspiciousness scores randomly.

QuixBugs: QuixBugs [6] is a collection of 40 bugs in simple algorithm implementations written in Java and Python. In addition to our validation set we also evaluate a subset of 27 Python bugs taken from QuixBugs. In particular, we exclude bugs that have no passing tests. Some tests cause timeouts

[2]https://lightgbm.readthedocs.io/en/stable/
[3]https://catboost.ai

or errors. Since our training data contains only passing or failing test runs (i.e., the program either outputs a correct or incorrect result) we simply treat such test outcomes as test failures. Including bugs with errors and timeouts in the training set might further improve performance on QuixBugs. Test executions are traced and processed in the same way as described earlier.

IV. Results

We find that our model considerably outperforms traditional SBFL metrics on our validation set and, by a smaller margin, also on QuixBugs (see Table I). Note that for QuixBugs, MAR and MFR are identical as the selected bugs only have a single buggy line. Results in Table I were obtained using all features described in Section III and a window size w of 3 (i.e., the preceding and the succeeding lines are included). We analyze the contribution of different feature types as well as the impact of the window size in Section IV-A.

Overall, MAR and MFR are lower (better) on QuixBugs. An explanation for this might be that the number of traced (i.e., executed) lines is much lower for the bugs in QuixBugs (average 7.6) than it is for bugs in RunBugRun (average 13.5). Also, bugs in QuixBugs only have a single buggy line. In contrast, for Top-N, our model performs significantly worse on QuixBugs; possibly because our the validation set's distribution is much closer to the training set than QuixBugs.

TABLE I
Evaluation results with a window size of $w = 3$

		MAR	**MFR**	**Top-1**	**Top-3**	**Top-5**
Validation Set	DStar^2	6.57	5.43	23.4%	49.4%	66.7%
	Ochiai	6.58	5.44	23.4%	49.3%	66.7%
	Ours	**3.69**	**2.66**	**56.4%**	**79.9%**	**88.2%**
	Random	7.27	5.97	16.9%	43.8%	62.3%
	Tarantula	6.61	5.45	23.6%	49.2%	66.1%
QuixBugs	DStar^2	3.70	3.70	**29.6 %**	55.6 %	**77.8 %**
	Ochiai	3.74	3.74	**29.6 %**	51.9 %	**77.8 %**
	Ours	**3.33**	**3.33**	**29.6 %**	**63.0 %**	74.1 %
	Random	4.04	4.04	7.4 %	51.9 %	**77.8 %**
	Tarantula	3.78	3.78	**29.6 %**	51.9 %	**77.8 %**

A. Ablation Study

An interesting question is to what extent different features contribute to the performance of our model. Here, we provide a brief ablation study. We first show performance impact of different groups of features (e.g., lexical, spectral etc.). Next, we also look at how training set and window size affect performance. Finally, we look at the most important features as reported by the LightGBM framework.

Feature Groups: We group features into four groups, namely, 1) spectral features, 2) SBFL formulae features, 3) control flow (in/out paths) features and 4) lexical features. Note that SBFL formulae, while also encoding spectral information, are placed in a separate group. Then, for each such group, we train and evaluate a classifier omitting features belonging to that group. Training and evaluation is repeated three times per group to obtain more stable results. As can be seen in Table II, including lexical features slightly lowers performance. Not surprisingly, omitting all spectral features, that is formulae score features as well as "direct" spectral features, greatly reduces performance. Still, path features are more important than either formulae or direct spectral features alone.

TABLE II
Results of ablation experiment.

	MAR	**MFR**	**Top-1**	**Top-3**	**Top-5**
No lexical feat.	**3.66**	**2.66**	**56.6 %**	**80.2 %**	88.5 %
All feat.	3.69	2.66	56.4 %	79.9 %	88.2 %
No formulae feat.	3.70	2.70	55.4 %	80.1 %	**89.0 %**
No spectral feat.	3.84	2.83	54.2 %	78.1 %	87.2 %
No path feat.	3.95	2.90	54.4 %	78.5 %	87.0 %
No spec.+formulae feat.	4.48	3.41	44.5 %	71.9 %	83.0 %

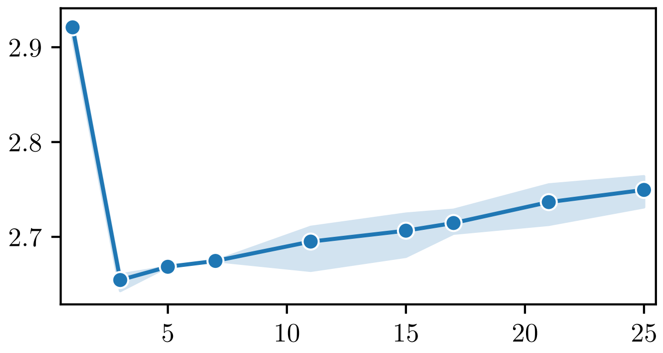

Fig. 1. MFR as a function of window size (x). The lowest MFR is obtained with a window size of three. Error bands show a 95% CI over three training runs.

Window size: Figure 1 shows how window size w affects performance on our validation set. We see a pronounced drop (i.e. improvement) in MFR for $w > 1$ and a steady performance decrease for $w > 3$. In other words, we obtain the best performance for $w = 3$, that is, a single context line preceding and following the focal line.

Training set size: We evaluate the effect of training set size by training with only a fraction of the available training data. As shown in Figure 2, MFR is decreasing (improving) steadily as the number of training samples is increased. While the curve starts to flatten out, we do not observe a saturation effect and more data could possibly further improve performance.

Feature Importance: Being based on decision trees, GBMs are to some degree *explainable*. In particular, LightGBM can report feature importance scores after a training run. Table III list the most important features by gain score. This score represents a value proportional to the decrease in training loss when using the corresponding feature for splitting. Note that due to multicollinearity in our data, care must be taken when interpreting these scores.

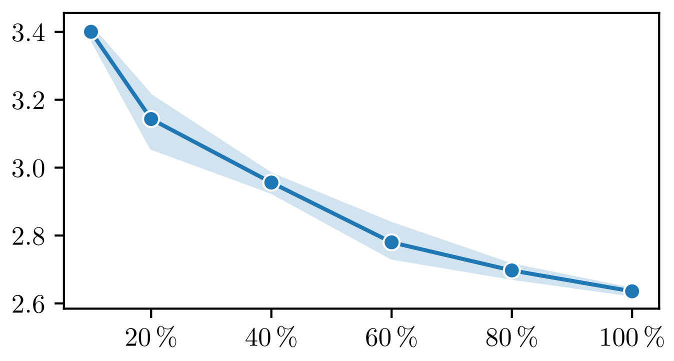

Fig. 2. MFR as a function of the training set size where 100 % uses the entire training data of roughly 22'000 samples (buggy programs). A window size is of $w = 3$ is used here. Error bands show a 95% CI over three training runs.

TABLE III
MOST IMPORTANT FEATURES BY GAIN ($w = 3$)

Feature	Gain Score	Rel.
`exec_pass_norm0`	49,147	1.00
`tarantula0`	25,623	0.52
`exec_pass_norm_max`	16,470	0.34
`exec_failed_norm_min`	16,412	0.33
`num_paths_out_fail1`	16,190	0.33

Still looking at table III, we see that the *normalized* number of line executions under passing tests (`exec_pass_norm0`) is reported as the most important feature. The zero at the end of the feature name indicates that this feature belongs to the focal line and not to a neighboring context line (i.e., features for the succeeding line would end in `1` and for the preceding line in `-1`). It is perhaps worth noting that the first *non*-normalized execution count feature only appears in 12th place (we only show top 5 in Table III) after 8 normalized versions of such features before it.

The Tarantula ranking score feature, again of the focal line, is listed next, followed by two more execution count features. The latter represent the minimum and maximum feature value inside the selected window and end with `min` and `max`, respectively. Last listed is `num_paths_out_fail1`, the number of outgoing paths of the line following the focal line under failing tests.

Of note, we can see that there is a wide margin between the first and second ranked features with the former being almost twice as important as the latter. Also, the ranking of the Tarantula feature should be taken with a grain of salt as there is a strong multicollinearity with the other two ranking score features (i.e., Ochiai and DStar).

TABLE IV
FEATURE IMPORTANCE BY WINDOW LEVEL ($w = 3$)

Level	Avg. Gain Score	Rel.
Max	7,248	1.00
Focal Line	6,190	0.85
Min	5,727	0.79
Succeeding Line	4,939	0.68
Preceding Line	3,868	0.53

Window level importance: As we have seen, features can have different positions or levels within the window. With a window size of $w = 3$, we have, beside the focal line, a preceding line and a succeeding line, as well as features for the minimum and maximum in the window. Grouping and averaging feature scores accordingly, we can estimate the importance of these levels. As can be seen from Table IV, taken together, maximum in-window features have the highest gain scores, followed by features of the focal line; features relating to the succeeding line are more important than those of the preceding line.

Summary of results: In short, our simple model performs significantly better than simple SBFL formulae. Including features of the preceding and succeeding lines improves performance, however, further increasing the window size leads to a gradual performance drop. Performance also depends on the number of training samples; even our relatively large training set does not seem to fully saturate our model. Ablation shows that spectral features are most important, but also path features seem to have significant impact. On the other hand, our simple lexical features seem to have a negative effect or, at best, no effect at all.

V. DISCUSSION

Line numbers can go a long way: As mentioned earlier, all of our features can be derived from execution traces. In particular, a sequence of executed line numbers is sufficient for the approach presented in this paper. Once obtained such as sequence, coverage and count spectra are trivial to calculate. We can even capture rudimentary control flow information simply by looking, for a specific line l, at its preceding (or alternatively, succeeding) line numbers. Computed features can be fit using efficient GBM models within minutes on modern consumer-grade hardware requiring no GPU.

Complementing features: Our ablation study shows that features of different types or from different spectra complement each other well. For instance, the top five most important features contain coverage, count spectra as well as control flow spectra. While this is not a new insight and many hybrid approaches have been proposed in the literature (e.g., DeepFL [12]), it seems worth noting that complementary features can be obtained from the same information source, namely simple execution traces with line numbers.

Normalization: Our results suggest that normalization is crucial for count spectra features. This is not surprising, as absolute counts vary widely across programs; normalization makes execution features somewhat comparable. We use a very simple normalization scheme, dividing by the total number of executions. Future work may investigate better ways of doing this normalization. Finally, normalization may also provide a way to incorporate time (or step count) features similar to "time percentages" devised by Yilmaz, Paradkar, and Williams [21].

Large windows don't help: While using windows greatly improves localization accuracy, somewhat surprisingly, windows larger than $w = 3$ have a detrimental effect on performance. More work is needed to understand why this is and at this point we can only offer speculation. For one, the optimal window size might be proportional to the lengths of programs. As the programs used in this work are very short, larger window might not take full effect. Also, windowing incurs padding and larger windows require more padding. Features in padded areas must be filled with dummy values (we use -1) which might pose some difficulty to our model.

Data is important: To our surprise, the curve in Figure 2 does not flatten out completely. This indicates that the available data is not fully saturating our model and that using more training data might yield further performance increases. This also emphasizes the importance of data in current and future fault localization systems. However, while we used small programs from programming contests, which are relatively easy to come by, it is not clear how thousands of execution traces can be obtained for real-world programs.

VI. Limitations & Threads to Validity

Bugs: Despite our utmost attention, we cannot fully exclude the possibility of bugs or other mistakes in our scripts or in our analysis.

Python only: So far, this work is limited to Python programs. Python was mainly chosen because of its `sys.settrace` feature which greatly facilitates tracing. Similar features can be found in other interpreted languages (e.g. Ruby[4]). However, for compiled languages, obtaining execution traces may be considerably more challenging and may require instrumenting byte code (e.g., JVM) or transforming intermediate representations (e.g., LLVM/Clang).

Real-world software: Our experiments use small programs from programming contests. Such programs are not representative of larger, real-world software projects, as for instance found in Defects4J [39]. Given that our results show a performance drop on QuixBugs (whose bugs are somewhat similar in style to RunBugRun), it is very unlikely that a model trained on such small programs would perform well on code of larger projects. For further experiments in this direction we would need execution traces of large, real-world projects which in turn would require a large dataset (e.g., $> 10,000$ samples) of executable bugs in real-world projects. Unfortunately, as of writing, we are not aware of any such dataset.

Baseline: We use simple SBFL formulae as baselines. While such formula are commonly used as baselines, much stronger FL systems have been devised in the literature. However, the two most commonly employed production-ready fault localization tools, GZoltar [40] and FLACOCO [41], both rely on the coverage spectrum and the Ochiai ranking formula and should thus not be very far from our baselines. Moreover, our work emphasizes on simplicity and a comparison with, say, a large language model or a system employing advanced program analysis would not be an apples-to-apples comparison. Finally, we could not find any detailed FL evaluation results on the Python version of QuixBugs in the literature.

[4]https://github.com/ruby/tracer?tab=readme-ov-file#linetracer

VII. Conclusions

In this work we presented a KISS (keep it simple and straightforward) approach to fault localization that leverages execution traces. We described simple ways to obtain features belonging to multiple spectra from such traces and how a machine learning model can be trained and used as a bug localizer. Our evaluation results show that our approach, although simple, clearly outperforms SBFL coverage formulae.

We hope that some of the presented ideas may inspire authors of existing FL tools. As we have shown, the step from pure coverage SBFL to a multi-spectral approach does not have to be very big. All that is necessary is the sequence of executed lines (e.g. line numbers). Even the transition from boolean coverage matrices to normalized count matrices may already lead to an appreciable improvement.

References

[1] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation of program spectra," *SIGPLAN Not.*, vol. 33, no. 7, pp. 83–90, Jul. 1998, ISSN: 0362-1340. DOI: 10.1145/277633.277647. (visited on 08/29/2024).

[2] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, 31:1–31:40, Oct. 2013, ISSN: 1049-331X. DOI: 10.1145/2522920.2522924. (visited on 08/30/2024).

[3] A. Ochiai, "Zoogeographical Studies on the Soleoid Fishes Found in Japan and its Neighbouring Regions-III," 1957. DOI: 10.2331/SUISAN.22.522.

[4] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02, New York, NY, USA: Association for Computing Machinery, May 2002, pp. 467–477, ISBN: 978-1-58113-472-8. DOI: 10.1145/581339.581397. (visited on 08/29/2024).

[5] J. A. Prenner and R. Robbes, *RunBugRun – An Executable Dataset for Automated Program Repair*, Apr. 2023. DOI: 10.48550/arXiv.2304.01102. arXiv: 2304.01102 [cs]. (visited on 07/25/2023).

[6] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH Companion 2017, New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 55–56, ISBN: 978-1-4503-5514-8. DOI: 10.1145/3135932.3135941. (visited on 04/21/2021).

[7] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar Method for Effective Software Fault Localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, Mar. 2014, ISSN: 1558-1721. DOI: 10.1109/TR.2013.2285319. (visited on 08/29/2024).

[8] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05, New York, NY, USA: Association for Computing Machinery, Nov. 2005, pp. 273–282, ISBN: 978-1-58113-993-8. DOI: 10.1145/1101908.1101949. (visited on 03/30/2021).

[9] J. Xuan and M. Monperrus, "Learning to Combine Multiple Ranking Metrics for Fault Localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sep. 2014, pp. 191–200. DOI: 10.1109/ICSME.2014.41. (visited on 08/30/2024).

[10] S. Yoo, "Evolving Human Competitive Spectra-Based Fault Localisation Techniques," in *Search Based Software Engineering*, G. Fraser and J. Teixeira de Souza, Eds., Berlin, Heidelberg: Springer, 2012, pp. 244–258, ISBN: 978-3-642-33119-0. DOI: 10.1007/978-3-642-33119-0_18.

[11] W. E. Wong and Y. Qi, "Bp neural network-based effective fault localization," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 04, pp. 573–597, Jun. 2009, ISSN: 0218-1940. DOI: 10.1142/S021819400900426X. (visited on 09/02/2024).

[12] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019, New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 169–180, ISBN: 978-1-4503-6224-5. DOI: 10.1145/3293882.3330574. (visited on 06/25/2024).

[13] Y. Li, S. Wang, and T. Nguyen, "Fault Localization with Code Coverage Representation Learning," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 661–673. DOI: 10.1109/ICSE43902.2021.00067. (visited on 09/01/2024).

[14] A. Z. H. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large Language Models for Test-Free Fault Localization," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24, New York, NY, USA: Association for Computing Machinery, Feb. 2024, pp. 1–12, ISBN: 9798400702174. DOI: 10.1145/3597503.3623342. (visited on 06/24/2024).

[15] S. Ji, S. Lee, C. Lee, H. Im, and Y.-S. Han, *Impact of Large Language Models of Code on Fault Localization*, Aug. 2024. DOI: 10.48550/arXiv.2408.09657. arXiv: 2408.09657 [cs]. (visited on 08/23/2024).

[16] M. N. Rafi, D. J. Kim, A. R. Chen, T.-H. (Chen, and S. Wang, "Towards Better Graph Neural Network-Based Fault Localization through Enhanced Code Representation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, 86:1937–86:1959, Jul. 2024. DOI: 10.1145/3660793. (visited on 08/30/2024).

[17] Y. Lou, Q. Zhu, J. Dong, *et al.*, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 664–676, ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3468580. (visited on 09/01/2024).

[18] Z. Zhang, Y. Lei, X. Mao, M. Yan, X. Xia, and D. Lo, "Context-Aware Neural Fault Localization," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3939–3954, Jul. 2023, ISSN: 1939-3520. DOI: 10.1109/TSE.2023.3279125. (visited on 09/02/2024).

[19] J. Qian, X. Ju, X. Chen, H. Shen, and Y. Shen, "AGFL: A Graph Convolutional Neural Network-Based Method for Fault Localization," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2021, pp. 672–680. DOI: 10.1109/QRS54544.2021.00077. (visited on 09/03/2024).

[20] J. Qian, X. Ju, and X. Chen, "GNet4FL: Effective fault localization via graph convolutional neural network," *Automated Software Engineering*, vol. 30, no. 2, p. 16, Apr. 2023, ISSN: 1573-7535. DOI: 10.1007/s10515-023-00383-z. (visited on 09/03/2024).

[21] C. Yilmaz, A. Paradkar, and C. Williams, "Time will tell: Fault localization using time spectra," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, New York, NY, USA: Association for Computing Machinery, May 2008, pp. 81–90, ISBN: 978-1-60558-079-1. DOI: 10.1145/1368088.1368100. (visited on 08/30/2024).

[22] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09, New York, NY, USA: Association for Computing Machinery, Aug. 2009, pp. 43–52, ISBN: 978-1-60558-001-2. DOI: 10.1145/1595696.1595705. (visited on 08/30/2024).

[23] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 56–66. DOI: 10.1109/ICSE.2009.5070508. (visited on 08/30/2024).

[24] R. Abreu, W. Mayer, M. Stumptner, and A. J. C. van Gemund, "Refining spectrum-based fault localization rankings," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09, New York, NY, USA: Association for Computing Machinery, Mar. 2009, pp. 409–414, ISBN: 978-1-60558-166-8. DOI: 10.1145/1529282.1529374. (visited on 08/30/2024).

[25] J. Sohn and S. Yoo, "FLUCCS: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017, New York, NY, USA: Association for Computing Machinery, Jul. 2017, pp. 273–283, ISBN: 978-1-4503-5076-1. DOI: 10.1145/3092703.3092717. (visited on 08/30/2024).

[26] Q. I. Sarhan and Á. Beszédes, "A Survey of Challenges in Spectrum-Based Software Fault Localization," *IEEE Access*, vol. 10, pp. 10 618–10 639, 2022, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3144079. (visited on 08/30/2024).

[27] L. Zhao, L. Wang, and X. Yin, "Context-Aware Fault Localization via Control Flow Analysis," *Journal of Software*, vol. 6, no. 10, pp. 1977–1984, Oct. 2011, ISSN: 1796-217X. DOI: 10.4304/jsw.6.10.1977-1984. (visited on 08/30/2024).

[28] G. Laghari and S. Demeyer, "On the use of sequence mining within spectrum based fault localisation," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, ser. SAC '18, New York, NY, USA: Association for Computing Machinery, Apr. 2018, pp. 1916–1924, ISBN: 978-1-4503-5191-1. DOI: 10.1145/3167132.3167337. (visited on 08/30/2024).

[29] Z. Zhang, Y. Lei, X. Mao, M. Yan, L. Xu, and J. Wen, "Improving deep-learning-based fault localization with resampling," *Journal of Software: Evolution and Process*, vol. 33, no. 3, e2312, 2021, ISSN: 2047-7481. DOI: 10.1002/smr.2312. (visited on 09/03/2024).

[30] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with AMPLE," in *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, ser. AADEBUG'05, New York, NY, USA: Association for Computing Machinery, Sep. 2005, pp. 99–104, ISBN: 978-1-59593-050-7. DOI: 10.1145/1085130.1085143. (visited on 09/11/2024).

[31] G. Laghari, A. Murgia, and S. Demeyer, "Localising faults in test execution traces," in *Proceedings of the 14th International Workshop on Principles of Software Evolution*, ser. IWPSE 2015, New York, NY, USA: Association for Computing Machinery, Aug. 2015, pp. 1–8, ISBN: 978-1-4503-3816-5. DOI: 10.1145/2804360.2804361. (visited on 09/11/2024).

[32] H. He, J. Ren, G. Zhao, and H. He, "Enhancing Spectrum-Based Fault Localization Using Fault Influence Propagation," *IEEE Access*, vol. 8, pp. 18 497–18 513, 2020, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2965139. (visited on 08/30/2024).

[33] M. Zeng, Y. Wu, Z. Ye, Y. Xiong, X. Zhang, and L. Zhang, "Fault localization via efficient probabilistic modeling of program semantics," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 958–969, ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510073. (visited on 09/11/2024).

[34] R. Puri, D. S. Kung, G. Janssen, *et al.*, *CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks*, Aug. 2021. DOI: 10.48550/arXiv.2105.12655. arXiv: 2105.12655 [cs]. (visited on 09/05/2022).

[35] J. A. Prenner and R. Robbes, "Out of Context: How important is Local Context in Neural Program Repair?" In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24, New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–13, ISBN: 9798400702174. DOI: 10.1145/3597503.3639086. (visited on 09/03/2024).

[36] C. J. Burges, "From RankNet to LambdaRank to LambdaMART: An Overview," (visited on 09/03/2024).

[37] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard, "A study of the behavior of several methods for balancing machine learning training data," *SIGKDD Explor. Newsl.*, vol. 6, no. 1, pp. 20–29, Jun. 2004, ISSN: 1931-0145. DOI: 10.1145/1007730.1007735. (visited on 09/04/2024).

[38] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017. [Online]. Available: http://jmlr.org/papers/v18/16-365.html.

[39] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014, New York, NY, USA: Association

for Computing Machinery, Jul. 2014, pp. 437–440, ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2628055. (visited on 04/08/2021).

[40] J. Campos, A. Riboira, A. Perez, and R. Abreu, "GZoltar: An eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012, New York, NY, USA: Association for Computing Machinery, Sep. 2012, pp. 378–381, ISBN: 978-1-4503-1204-2. DOI: 10.1145/2351676.2351752. (visited on 09/22/2021).

[41] A. Silva, M. Martinez, B. Danglot, D. Ginelli, and M. Monperrus, *FLACOCO: Fault Localization for Java based on Industry-grade Coverage*, Mar. 2023. DOI: 10.48550/arXiv.2111.12513. arXiv: 2111.12513 [cs]. (visited on 09/02/2024).