Revisiting Unnaturalness for Automated Program Repair in the Era of Large Language Models

Aidan Z.H. Yang
Carnegie Mellon University
Pittsburgh, PA, United States
aidan@cmu.edu

Sophia Kolak
Carnegie Mellon University
Pittsburgh, PA, United States
sdkolak@andrew.cmu.edu

Vincent Hellendoorn
Carnegie Mellon University
Pittsburgh, PA, United States
vhellendoorn@cmu.edu

Ruben Martins
Carnegie Mellon University
Pittsburgh, PA, United States
rubenm@andrew.cmu.edu

Claire Le Goues
Carnegie Mellon University
Pittsburgh, PA, United States
clegoues@cs.cmu.edu

2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE) | 979-8-3315-0569-1/25/$31.00 ©2025 IEEE | DOI: 10.1109/ICSE55347.2025.00089

***Abstract*—The problem of software quality has motivated the development of a variety of techniques for Automatic Program Repair (APR). Meanwhile, recent advances in AI and Large Language Models (LLMs) have produced orders of magnitude performance improvements over previous code generation techniques, affording promising opportunities for program repair and its constituent subproblems (e.g., fault localization, patch generation). Because models are trained on large volumes of code in which defects are relatively rare, they tend to both simultaneously perceive faulty code as unlikely (or "unnatural") and to produce generally correct code (which is more "natural"). This paper comprehensively revisits the idea of (un)naturalness for program repair. We argue that, fundamentally, LLMs can only go so far on their own in reasoning about and fixing buggy code. This motivates the incorporation of traditional tools, which compress useful contextual and analysis information, as a complement to LLMs for repair. We interrogate the role of entropy at every stage of traditional repair, and show that it is indeed usefully complementary to classic techniques. We show that combining measures of naturalness with class Spectrum-Based Fault Localization (SBFL) approaches improves Top-5 scoring by 50% over SBFL alone. We show that entropy delta, or change in entropy induced by a candidate patch, can improve patch generation efficiency by 24 test suite executions per repair, on average, on our dataset. Finally, we show compelling results that entropy delta for patch classification is highly effective at distinguishing correct from overfitting patches. Overall, our results suggest that LLMs can effectively complement classic techniques for analysis and transformation, producing more efficient and effective automated repair techniques overall.**

***Index Terms*—Program Repair, Deep Learning, Large Language Models**

I. Introduction

The problem of software quality has motivated the development of a variety of techniques for Automatic Program Repair (APR) [1]–[5]. At a high level, dynamic APR approaches use test cases to define a defect to be repaired and functionality to retain, and to localize the defect to a smaller set of program lines. APR techniques generate candidate patches in a variety of ways, such as by heuristically instantiating pre-defined repair template [1], [6], or by customizing symbolic techniques to synthesize new code [5], [7].

Meanwhile, recent advances in AI, including but by no means limited to advances in Transformer [8] based language models, have produced orders of magnitude performance improvements over previous ML techniques for code generation [9], [10]. AI therefore affords promising opportunities for program repair [2], [3], [11]–[13] and its constituent subproblems, like fault localization [14]. The applicability of language models to repair makes sense: models are trained on large volumes of code in which defects are relatively rare. Since their training objective encourages next-token prediction, well-trained language models tend to simultaneously perceive faulty code as unlikely (or "unnatural") and to produce code that is correct, as correct code is more "natural" [15].

Indeed, the naturalness of code and unnaturalness of buggy code is now a well-established phenomenon [15], [16]. However, the bulk of existing research on this topic relies on relatively simple n-gram language models [17]. Compared to present-day LLMs, these models provide a very poor estimator of code predictability. The "unnaturalness" of buggy lines has therefore, to date, been useful primarily as an explanatory metric, showing limited utility for precise defect localization, let alone repair. With the development of larger and more sophisticated LLMs, model perplexities have decreased by multiple orders of magnitude. And, indeed, LLMs also tend to score buggy code as less "natural" than correct code [18]. LLMs are therefore more accurate adjuncts both for estimating naturalness, and for fault localization or correct patch identification [19], [20].

In this paper, we revisit the idea of (un)naturalness for program repair. The fundamental idea behind using an LM alone — even a hypothetically optimal one — for repair treats predictability as ultimately equivalent to correctness. This assumption is specious: LLMs adopt preferences based on a corpus with respect to training loss that rewards imitation. Beyond the fact that LLMs necessarily train on buggy code,

This work was partially supported by the Future Enterprise Security Initiative at Carnegie Mellon CyLab (FutureEnterprise@CyLab).

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

LLMs generate and score text one token at the time. Given that, they may well prefer a subtly incorrect implementation spread across several readable lines over a correct but difficult-to-understand one-line solution, as the per-token surprisal of the former may be strictly lower than the latter. Judgment of code correctness requires substantially more context than an LLM has access to including, but not limited to, test cases test behavior, and developer intent. Although some of this information could be provided as context, it will lie outside the training distribution.

This implies that LLMs can only go so far on their own in reasoning about and fixing buggy code. It moreover motivates the use of traditional tools, which compress such information, as a complement to LLMs in repair, which has indeed shown promising recent results for the patch generation stage in particular [19] (acknowledging the risk of training data leakage in any such experiment [21]).

We go beyond prior work by interrogating the role of entropy as a complement to traditional repair at every stage:

Fault localization (FL). End-to-end dynamic APR relies on fault localization to narrow a bug to a smaller set of source locations. Improving fault localization accuracy is key to improving repair efficiency [22], [23]. Although FL accuracy is improving, both commonly-used [24] and state-of-the-art ML-based techniques still suffer from the tendency to assign the same FL score to large amounts of code. For example, we find that, on the DEFECTS4J dataset, Ochiai SBFL [24] and TransferFL [23] produce an average of 2.9 and 0.96 tied lines per bug, respectively.

We show that by incorporating entropy into fault localization, the variance of suspiciousness scores increase by 87% for Ochiai SBFL, and overall accuracy increases as well (i.e., the Top-5 score improves from 94 to 145).

Plausible patch generation. APR approaches typically generate multiple potential code changes in search of *plausible* patches that cause the program to pass all tests. Executing tests (and to some extent, compiling programs to be tested) dominates repair time: the template-based approach TBAR [1] spends approximately 2% of its total time creating patch templates, 6% generating patches from templates, and 92% running tests on generated patches. Regardless of the patch generation method (e.g., symbolic techniques [4], [5], [7], template instantiation [1], [6], or machine learning models [19]) repair *efficiency* is best approximated in terms of the number of patches that must be evaluated to find a (good) repair [25].

We show that entropy, when used to order candidate patches for evaluation, can improve the efficiency of generic template-based repair by 24 tested patches per bug, on average.

Patch correctness assessment. Plausible patches are not always correct, in that they can fail to generalize to the desired behavior beyond what is tested in the provided test suite [26]. Some approaches address this problem via a post-processing step that identifies (and filters) plausible-but-incorrect patches, typically by combining program analysis and machine learning [20], [27], [28]. However, techniques to date are typically trained on the same test suites used for patch generation, imposing a project-specific training burden (and an expensive one, when dynamic signals are required), and posing a significant risk of overfitting [20], [26].

We show that entropy can rank correct patches 49% more effectively (in Top-1) than state-of-the-art patch ranker Shibboleth [28], without using any project-specific training data. We also show that entropy is highly effective in disambiguating correct from overfitting patches produced by ITER [29], a modern ML-based tool.

In summary, we make the following contributions.

- **Combination of entropy with prior FL approaches.** We propose and evaluate a combination of an LLM's measure of entropy with previous approaches for FL. We show for the first time that entropy and non-LLM-based approaches are highly complementary, and that their combination is most effective.
- **Entropy delta for efficient template-based patch generation.** We introduce entropy delta as a patch-naturalness measure to rank patches before running tests. We show that entropy-delta can be used to effectively filter test-failing patches, and on average reduces running tests by 24 patches per bug in our dataset. We combine entropy delta patch ranking with a prior template-based program repair technique, TBAR [1], and release a more efficient version of TBAR for future research.
- **Entropy delta for patch correctness assessment.** Using the largest and most recent source-level dataset for plausible/correct patch ranking and classification, we show that entropy delta can distinguish correct from overfitting patches with higher precision and accuracy than prior work [28]. We also show that entropy delta can also be used with state-of-the-art ML-based APR tools [29] to improve patch correctness assessment.
- **Artifact availability**. Our data, tool, and results are available and are released as open-source.[1]

II. ILLUSTRATIVE EXAMPLE

Consider the buggy and fixed versions of (Chart, 4) from DEFECTS4J [30], shown in Figure 1a. The original buggy code is missing a null check, which the developer fixed by adding `if(r != null)` around the implicated code at line 4493.

The TBAR [1] template-based program repair technique produces candidate patches by repeatedly instantiating applicable templates at program statements, ordered by Ochiai SBFL suspiciousness score. For example, Figure 1b a TBAR-generated patch that does not cause the tests to pass, and so is discarded, and the search continues. Given Chart's associated test suite, the Ochiai SBFL approach [22] assigns line 4473 the highest suspicious score in Chart of 0.52; line 4493, a suspiciousness score of 0.49; and 0.03 to lines 4494 and onward. Using only SBFL for fault localization ranking, the actual faulty line at line 4493 is ranked as the 10th most

[1]https://github.com/squaresLab/entropy-apr-replication

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

```
4473  XYItemRenderer r = getRendererForDataset(d);
                                      //SBFL=0.52, E=0.48
4474       ...
4493 - Collection c = getRenderer().getAnnotations();
                                      //SBFL=0.49, E=1.59
4494 + if (r != null) {
4495 +    Collection c = r.getAnnotations();
                                      //E=1.34
4496   ...
4502 + }
```

(a) Chart 4 buggy code and developer fix.

```
4493 - Collection c = getRenderer().getAnnotations();
                                      //E=1.59
4494 + if (r == null) {
4495 +    return null;            //E=2.77
4496 + }
```

(b) Chart 4 buggy code and test failing TBAR patch #1.

```
4493 - Collection c = getRenderer().getAnnotations();
                                      //E=1.59
4494 + if (r == null) {
4495 +    continue;               //E=1.98
4496 + }
```

(c) Chart 4 buggy code and test passing TBAR patch #19.

Fig. 1: Chart bug 4 from DEFECTS4J with its developer-written fix, a test-failing patch generated by TBAR, and a test-passing patch generated by TBAR. We show the InCoder-produced entropy of code in each patch.

suspicious. This does not prevent TBAR from considering it, but does cost time.

TBAR can produce patches that pass all tests for this bug, such as the one shown in Figure 1c. In the interest of reasoning about efficiency, we hold fault localization constant [25]; given that, this is the 19th patch attempted. However, although this patch prevents the null pointer exception, it does not generalize beyond the provided tests to capture the apparent developer intent. Importantly, TBAR *can* produce the correct patch (from Figure 1a), if configured to execute beyond the first test-passing patch found — it is the 70th patch attempted, but only the second that passes all tests.[2]

LLM-based entropy provides useful clues, here, however. First, consider fault location: we use InCoder [31],[3] to measure the entropy of every line in this file. Rank-ordering them, line 4473 is ranked 8th-most-surprising. This is better than the SBFL technique on face. However, their real utility appears to lie in combination: re-ranking the lines receiving the Top-10 SBFL suspicious scores by InCoder entropy values, puts line 4493 at rank 2. We investigate how entropy in conjunction with SBFL performs for fault localization across multiple bugs and projects, as well as how different LLMs affect its performance.

[2]Using SBFL fault localization, TBAR produces these patches at 1076 and 1127, respectively.

[3]When prompted with the code and asked to fix the bug directly, InCoder does not produce a test-passing patch in few-shot setting. Note that GPT4 fixes the bug correctly, and reports the git commit associated with the fix, implicating data leakage.

We can also measure the naturalness of generated patches, such as by calculating the change in entropy, which we call entropy delta ($\triangle E$), between the original buggy line of code and the proposed patches. The $\triangle E$ for the test-failing patch is -0.39; for the test-passing but still-incorrect patch is -1.18; and for the correct patch is 0.25.

The entropy delta scores do not perfectly predict behavior (note that the test-failing patch has a higher score than the test-passing-but-incorrect patch), but it still suggests:

1) Entropy delta can improve efficiency by suggesting the order to test patches. Test execution time is the dominant cost in program repair. By using entropy to rank potential patches *before* testing them, to suggest the order in which to do so, both test-passing patches can be found within 6 attempts (improving on 19 to the first test-passing in the default mode, and 90 to find the second, correct patch).
2) Entropy delta can potentially help disambiguate plausible-but-incorrect from genuinely correct patches.

We evaluate these relationships in detail in the rest of this work, showing how entropy can usefully complement traditional approaches to automatic localization and transformation in the context of program repair.

III. Approach

We ask and answer the following research questions about LLM entropy for APR:

- **RQ1: How can entropy improve fault localization?** We empirically evaluate established fault localization technique, and observe whether and how they benefit from the use of entropy scores.
- **RQ2: How can entropy improve patch generation efficiency?** To measure how entropy can be used for patch generation efficiency, we use it to rank proposed patches generated by an APR technique before running tests.
- **RQ3: How effectively does an entropy delta identify correct patches?** We investigate the degree to which entropy deltas can differentiate plausible patches (patches that passes all tests) and correct patches (patches that correctly fix the bug). We investigate both classification (i.e., labelling patches individually as correct or not) and ranking (i.e., ranking patches by likelihood of correctness).

In this work, given an LLM model, we calculate the entropy of a line of code as follows: $Entropy = -\sum_{i=1}^{n} \frac{log(p_{t_i})}{n}$. Where $log(p_{t_i})$ is the model probability of generating token t_i given the context and previously generated tokens, and n is the number of tokens in a potentially buggy line or plausible patch.

This section describes how we use entropy for fault localization (Section III-A); our development of entropy delta for patch evaluation (Section III-B); and our modifications to TBAR to enable our study of improved patch efficiency (Section III-C). The next section (Section IV) describes datasets and metrics.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

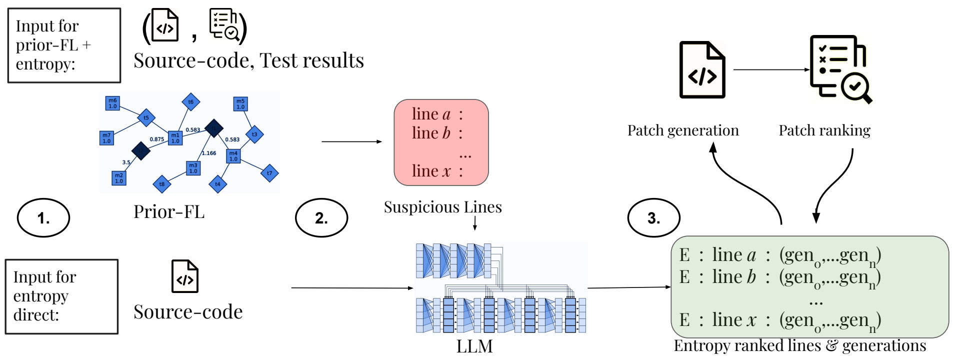

Input for prior-FL + entropy:
Source-code, Test results
Prior-FL
1.
Input for entropy direct:
Source-code
2.
line a :
line b :
...
line x :
Suspicious Lines
LLM
3.
Patch generation
Patch ranking
E : line a : (gen_o,...gen_n)
E : line b : (gen_o,...gen_n)
...
E : line x : (gen_o,...gen_n)
Entropy ranked lines & generations

Fig. 2: Fault localization pipeline using entropy. (1) We take a prior FL suspicious score list, (2) query each code line for entropy values, and (3) re-rank the list using LLM entropy scores.

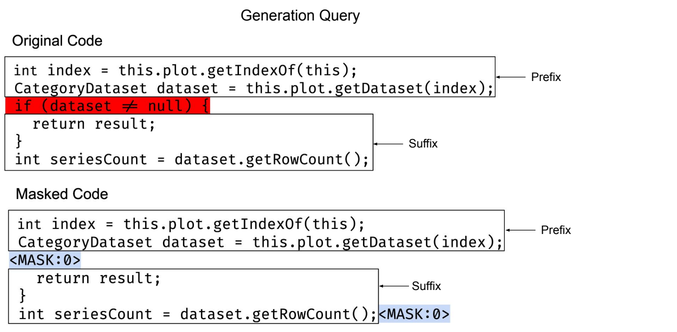
A. Entropy for fault localization

We integrate raw entropy scores into prior fault localization techniques. Figure 2 overviews the approach. We begin with suspiciousness scores provided by a prior FL tool for a given file. We focus on top-N identified lines, as developers do not typically inspect more than 5 candidates [32]; this is consistent with prior work. We choose 6 and 10 for N, and name the two approaches 6-filter and 10-filter, as these are two quantities near the Top-5 that can still significantly impact Top-5 scores. The 6- and 10-filters use the prior FL tool as a first pass filter before applying entropy to provide finer-grained ranking. We experimented with several cutoffs, but observed that reranking all code lines in a file degrades performance compared to focusing on a more-likely subset. Test-based FL often successfully identifies suspicious code blocks, but produce ties on statements within blocks; ML-based approaches judge each line across entire files, a large decision space, and thus are noisy.

Next, we query an LLM for entropy scores for each line of code in each implicated file. We tokenize the entire file, iteratively masking each line, and querying the model for each line's entropy. We used a sliding context window with 2048 tokens (i.e., the maximum attention window of our smallest selected LLM) surrounding the mask as suffix and prefix context. This allows us to assign entropy-based scores for all code in a file, even those longer than the LLM's context window.

We then re-rank the filtered (top-N) suspicious code identified by the prior technique by entropy score, and validate the ranked list according to the actual fault location in the dataset. We did not observe ties between re-ranked lines in our experiments. Even duplicate lines are unlikely to be identically "surprising" in this setting, because the prefix/suffix context must also be identical. We did not encounter such ties, rounding entropy scores to 4 places.

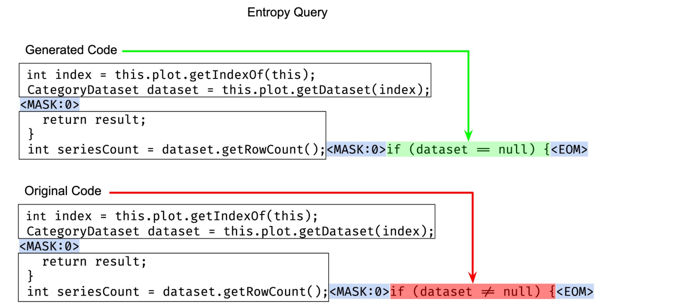
We incorporate entropy into three previous FL techniques: SBFL using the Ochiai formula [22], TransferFL [23] and LLMAO [14]. Ochiai is a common formula in SBFL used in traditional APR practices (e.g., TBar). TransferFL and LLMAO are the current state-of-the-art FL techniques using transfer-learning and LLMs, respectively.



```
int index = this.plot.getIndexOf(this);
CategoryDataset dataset = this.plot.getDataset(index);
if (dataset ≠ null) {
  return result;
}
int seriesCount = dataset.getRowCount();
```

```
int index = this.plot.getIndexOf(this);
CategoryDataset dataset = this.plot.getDataset(index);
<MASK:0>
  return result;
}
int seriesCount = dataset.getRowCount();<MASK:0>
```

(a) An example of entropy delta query from a code line deletion patch. The entropy delta value of the deleted line is the difference between the original line and a blank line.



```
int index = this.plot.getIndexOf(this);
CategoryDataset dataset = this.plot.getDataset(index);
<MASK:0>
  return result;
}
int seriesCount = dataset.getRowCount();<MASK:0>if (dataset == null) {<EOM>
```

```
int index = this.plot.getIndexOf(this);
CategoryDataset dataset = this.plot.getDataset(index);
<MASK:0>
  return result;
}
int seriesCount = dataset.getRowCount();<MASK:0>if (dataset ≠ null) {<EOM>
```

(b) An example of entropy delta query from a code line replacement patch. The entropy delta value of the replaced line is the difference between the original line and the replacement line.

Fig. 3: Example entropy delta queries from an LLM. The MASK tokens enable models to learn the contextual relationships between tokens and make entropy predictions for missing, new, or replacement tokens. The EOM token is a special token that indicates the end of a mask.

B. Entropy-Delta

To evaluate patch naturalness, which we use in both patch prioritization during generation/evaluation and patch correctness prediction, we introduce the concept of an "entropy delta". The entropy delta describes how code replacement

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

changes the naturalness of a block of code. Figure 3a and Figure 3b give examples for our usage of entropy delta for ranking patches. Figure 3a shows the process of masking out a deleted line of code and querying the LLM for the change in entropy using that mask (i.e., the change in entropy without the original line). Figure 3b shows the process of querying the LLM for the change in entropy if the tokens of the original line of code is replaced with new tokens of patch code. If the patch is an insertion of a blank new line, we query the entropy delta between the "newline" token and the original line of code. For the case of an insertion, we measure the entropy delta between the new code line and the original blank line.

An entropy delta is the change in entropy before and after a line in code is replaced. For instance, if the line's original entropy is 1.0, and the replacement line's entropy is 0.5, then the line has an entropy delta of +0.5, as in, replacing that line lowered entropy by 0.5. A significant reduction in entropy (large, positive entropy delta) means that the replacement code lowered the entropy, implying both that the original statement may have been buggy and that the patch is more natural for that region of code. A negative entropy delta means that the replacement code increased entropy; the patch is less natural at that location. An entropy delta of 0 means that the patch is as natural as the original code.

C. Modified TBar

Our patch efficiency experiments ask how entropy can speed improve patch generation efficiency. We evaluate it primarily in the context of TBAR [1], the best-performing prior template-based program repair technique. We avoid focusing on ML-based APR techniques (even though many outperform TBAR [19], [23], [33]) because our goal is a controlled evaluation of entropy *without* learned patterns. Evaluating based on a technique that otherwise also relies on trained ML models fails to isolate the effect of entropy per se.

TBAR is a template-based APR approach integrated with DEFECTS4J V1.2. We modify it in several ways First, we enable TBAR to continue seeking patches after the first test-patching patch is found. Second, we enable TBAR to either generate patches, or evaluate them in a customized order (such as one provided by an entropy delta ranking). These changes are necessary primarily to evaluate the effect that incorporating entropy measurements can have on APR efficiency, which entails evaluating potential patches in an alternative order. Our TBAR extension also includes refactoring for modifiability/extensibility, as well as a more accurate patch caching mechanism (caching the patched source, rather than the patch alone) to improve experimental efficiency.

We provide the modified code with our replication package.[4]

IV. METRICS AND DATSETS

In this section, we describe the models we use for entropy (Section IV-A), evaluation metrics (Section IV-B), and bug and patch datasets considered (Section IV-C) in our experiments.

[4]https://github.com/squaresLab/TBar

A. LLMs

We used InCoder [31], Starcoder [34], and Code-Llama2 [35]. The three LLMs were trained on open-source code and are capable of infilling with bidirectional context. The InCoder model [31] is a large-scale decoder-only Transformer model with 6.7 billion (6.7B) parameters. The model was trained on a dataset of code from public open-source repositories on GitHub and GitLab, as well as from StackOverflow. InCoder was primarily trained for code infilling, which involves the insertion of missing code snippets in existing code, using a causal-masked objective during training. However, its versatility enables it to be utilized in a variety of software engineering tasks, including automatic program repair. Starcoder and Llama-2 were trained with a similar autoregressive plus causal-mask objective as InCoder. Starcoder was trained with 15.5B parameters. Code-Llama2 has three versions available: 7B, 13B and 34B. We choose the 7B version as it is the closest in size to InCoder. Although the three LLMs were not specifically trained for repair, their large architectures and training objectives could imply that their entropy values on a particular region of code could suggest code naturalness. We use an LLM temperature of 0.5 for all experiments. Higher temperatures produce more randomness in LLM output. In prior APR work, this generates more diverse plausible patches [19]. However, higher temperature lead an LLM to find more tokens "unsurprising", increasing false positives in our setting (i.e., too many patches considered unsurprising, and therefore correct). We swept from 0.5 to 0.9, and found 0.5 to yield the best results.

B. Metrics

Fault localization. We measure fault localization effectiveness by counting the number of known faulty lines that appear in the Top-N position of a ranked list. The Top-N measure has been widely used in both APR and fault localization research as a standard measure [36]. Existing studies [32] showed that over 70% of developers inspect only the Top-5 suggested elements. We use Top-5, Top-3, and Top-1 for fault localization ranking.

Patch generation efficiency. We measure the effect of reranking generated potential patches in terms of the number of patch evaluations saved by doing so. Patch evaluations are established as a hardware- and program-independent measure for APR efficiency [25], and a proxy for compute time.

Patch correctness. The patch correctness literature has considered two related but separate tasks: patch *ranking*, and patch *classification*. *Ranking* looks at the accuracy of a technique in ordering or prioritizing a set of potentially-correct patches; *classification* looks at technique accuracy at a binary decision task (whether a plausible patch is, or is not, correct). We evaluate entropy-delta's effectiveness in both settings.

For patch ranking, we rank all plausible patches of a bug by their entropy-delta values, and evaluate our ranking by Top-N scores (cf. fault localization, above). We use Top-1 and Top-2 for patch ranking following Ghanbari et al. [28], as some bugs in our dataset only have 2 plausible patches available.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

TABLE I: DEFECTS4J bugs with at least one patch passing tests (RQ2), and a developer fix (RQ3). Projects denoted by † are part of the ITER replication package only (RQ3).

	DEFECTS4J V1.2 #bugs Patch efficiency (RQ2)		**DEFECTS4J V2.0 #bugs** Patch correctness (RQ3)	
	Incl.	Total	Incl.	Total
Chart	11	26	19	26
Closure	19	133	64	174
Lang	14	65	35	64
Math	21	106	67	106
Mockito	3	38	1	38
Time	4	27	11	26
Cli†	-	-	6	39
Codec†	-	-	3	18
Compress†	-	-	4	47
Csv†	-	-	2	16
JacksonCore†	-	-	3	26
Total	72	395	215	580

For patch classification, we convert entropy-delta values into binary labels. We label patches with a positive entropy-delta as "more natural" (i.e., more likely to be correct), and patches with a negative entropy-delta "less natural" (i.e., less likely to be correct). To measure entropy's ability to isolate correct and incorrect patches, we use +recall and -recall. +Recall measures to what extent correct patches are identified, while -recall measures to what extent incorrect patches are filtered out. We use accuracy, precision, and F1 scores to assess classification effectiveness over the entire dataset.

C. Datasets

We use the DEFECTS4J [30] dataset as the basis of our experiments. DEFECTS4J is a well-established set of documented historical bugs in Java programs with associated tests and developer patches. It is commonly used in APR, testing, and fault localization research. However, each research question requires a different subset of the dataset. Table I shows the number of bugs in each project that have at least one plausible patch produced by TBAR (for analyzing patch efficiency) and a developer fix (for analyzing patch correctness) along with plausible but incorrect patches, as well as the subsets of the defects we consider in each research question.

1) Fault localization: We used DEFECTS4J V1.2 for the fault localization experiments, as required for compatibility with the considered prior fault localization tools' replication packages. Actual repairability is not relevant to evaluating fault localization accuracy that is, we can evaluate how effectively a fault is localized regardless of whether any APR technique can fix that fault. We are therefore able to consider all 395 bugs in DEFECTS4J V1.2 in these experiments.

2) Efficiency: Like prior fault localization tools, TBAR's replication package and codebase is only compatible with DEFECTS4J V1.2. Additionally, for patch generation, the experimental goal is to evaluate the degree to which entropy can improve repair *efficiency*, or how many candidate patches must be considered before a repair is found. These experiments must therefore consider bugs that vanilla TBAR can actually historically repair; entropy can only improve efficiency for fixable bugs. We therefore focus on the subset of 72 DEFECTS4J V1.2 bugs on which vanilla TBAR succeeds at finding a plausible patch at least once within 3 hours.

3) Correctness: Most of our patch correctness evaluation is predicated on curated datasets from the replication packages for Shibboleth [28] and Panther [27]. Shibboleth and Panther are tools that leverage static and dynamic heuristics from both test and source code to rank and classify (respectively) plausible patches. These tools' replications include large, curated datasets of patches generated by a wide array of prior APR techniques for bugs from DEFECTS4J V2.0. Each bug in a data set has one correct patch and several plausible (i.e., test passing) but incorrect patches; techniques can then be evaluated per their ability to identify the correct patches.

For patch correctness *ranking*, we use a dataset of 1,290 patches produced by 29 prior APR tools on DEFECTS4J V2.0 curated to evaluate Shibboleth [28]. For patch *classification*, we use a dataset of 2,147 patches generated by 16 prior APR tools on DEFECTS4J V2.0, curated to evaluate Panther [27].

The patches in these established datasets were produced by previous-generation APR techniques, and include only one ML-based approach, SequenceR [37], which uses sequence-to-sequence learning. To better assess the applicability of our findings to modern, state-of-the-art ML-based approaches, we also evaluate how well entropy differentiates overfitting and correct patches generated by ITER [29]. ITER trains its own neural model and does not rely on an LLM; thus, while any ML model aims to optimize for some measure of statistical likelihood, ITER patches do not explicitly result from the models or type of models we use in entropy computation. The ITER replication package provides 79 pairings of labelled correct and plausible-but-incorrect patches generated on defects from DEFECTS4J V2.0.

We consider the dataset from Shibboleth and Panther, which corresponds to 6 out of 17 DEFECTS4J V2.0's total projects. Additionally, we also consider ITER's dataset which includes 5 more projects from DEFECTS4J V2.0's. The additional 5 projects from ITER are indicated via † in Table I.

V. RESULTS

We present results on our three primary research questions, addressing fault localization (Section V-A), patch generation efficiency (Section V-B), and patch correctness assessment (Section V-C).

A. RQ1: Fault Localization

In this research question, we compared 24 different configurations for fault localization. Our analysis aims to determine the most effective approach for identifying the buggy statement in a series of one-line bugs. We first measure entropy directly for fault localization with our three selected LLMs: Code-Llama2, Starcoder, and InCoder. We then measure the fault localization accuracy of three prior fault localization tools: SBFL [22], TransferFL [23], and LLMAO [14]. Finally, we use entropy to re-rank prior fault localization tools and observe that entropy re-ranking largely improves prior tools. Table II

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Top-N scores on 395 bugs from DEFECTS4J V1.2.0 from 3 prior tools and re-ranking with entropy from three pre-trained LLMs: InCoder (6B), LLAMA-2 (7B), and Starcoder (15.5B)

FL type	re-rank Filter	Technique	Top-1	Top-3	Top-5
Entropy		entropy-Llama2	5	20	41
		entropy-Starcoder	9	35	55
		entropy-InCoder	**38**	**90**	**116**
SBFL		SBFL	24	61	94
	10-filter	entropy-Llama2	15	37	84
		entropy-Starcoder	15	40	88
		entropy-InCoder	50	98	133
	6-filter	entropy-Llama2	25	84	**145**
		entropy-Starcoder	28	86	144
		entropy-InCoder	**55**	**117**	141
TransferFL		TransferFL	**69**	126	145
	10-filter	entropy-Llama2	33	82	105
		entropy-Starcoder	39	92	126
		entropy-InCoder	49	114	144
	6-filter	entropy-Llama2	38	131	**184**
		entropy-Starcoder	44	138	178
		entropy-InCoder	57	**156**	182
LLMAO		LLMAO	**87**	134	**149**
	10-filter	entropy-Llama2	38	131	145
		entropy-Starcoder	45	107	144
		entropy-InCoder	77	131	146
	6-filter	entropy-Llama2	49	121	143
		entropy-Starcoder	36	84	151
		entropy-InCoder	81	**142**	**151**

shows the Top-N scores (N = 1,3,5) on all configurations of our experiment. We observe that the entropy of InCoder, the smallest LLM in our lineup, is the most effective for fault localization. This is consistent with recent results [19] showing that InCoder, trained with an objective of predicting missing code from a bidirectional context, is more effective at program repair tasks than larger but purely causal generative LLMs.

SBFL. From Table II, we observe an overall decrease in Top-N scores using either Code-Llama2 or Starcoder entropy with a 10-filter. However, all Top-N scores improve with the 6-filter. In particular, the Top-3 score of 84 for Llama2 entropy improves upon SBFL by 38%, and the Top-3 score of 86 for Starcoder entropy improves upon SBFL by 41%. Using InCoder entropy to re-rank SBFL shows substantial improvements across all Top-N and the two types of filters. InCoder entropy-SBFL with a 10-filter achieves a Top-1 score of 50 (108% improvement), and 5-filter achieves a Top-1 score of 55 (129% improvement). Similarly, the Top-3 and Top-5 scores improve by 61% and 41%, respectively for InCoder entropy-SBFL with a 10-filter. The Top-3 and Top-5 scores improve by 92% and 50% respectively for the 10-filter.

TransferFL. As seen in Table II, we observe an improvement from entropy on TransferFL's Top-3 and Top-5 scores using a 6-filter. In particular, 6-filter InCoder-entropy with TransferFL Top-3 is 156 (24% improvement), and 6-filter Llama2-entropy with TransferFL Top-5 is 184 (27% improvement). However, 6-filter InCoder-entropy with TransferFL yields a Top-1 score of 57, which is a 17% decrease in performance than TransferFL by itself. As compared to state-of-the-art machine learning based FL techniques, we observe that entropy scores perform worse on Top-1.

TABLE III: entropy delta ranking scores of 72 plausible patches generated by TBAR per DEFECTS4J project. The mean rank decrease is 24 and the median is 5.

Project	Improves ranking	Lowers ranking
Chart	11	2
Closure	15	4
Lang	11	2
Math	16	3
Mockito	1	0
Time	3	1
Total	60	12

LLMAO. Similar to the results of TransferFL, re-ranking with entropy improves fault localization results using the 6-filter. Furthermore, only entropy calculated using InCoder shows an improvement over LLMAO alone for Top-3 and Top-5, with a 8% and 1% improvement, respectively. Since LLMAO is already an LLM-based FL tool, LLM entropy re-ranking provides relatively marginal improvements as compared to prior non-LLM-based FL tools. Additionally, LLMAO fine-tunes on CodeGen 16B [38], which is larger LLM than our three chosen LLMs. Our results show, however, that LLMAO and InCoder are complementary; having another LLM take a "second pass" is beneficial for fault localization even when an LLM is entailed in the first pass. We also note that while Top-1 scores are most important for human debugging, APR can typically attempt more repair locations, and thus improvement to top-3 and top-5 can also be beneficial in this context [39].

Overall, SBFL benefits the most with InCoder's entropy re-ranking. SBFL produces the most ties in suspiciousness scores (2.9 ties per bug on average); the additional information provided by entropy values helps to break these ties. TransferFL and LLMAO benefit from entropy re-ranking primarily when using a 6-filter. These findings suggest that incorporating entropy as a heuristic in fault localization can improve the accuracy of identifying the buggy statement, particularly but not exclusively when used in conjunction with SBFL.

RQ1 Summary

We leverage entropy for fault localization in DEFECTS4J programs and show that, while entropy alone is only somewhat effective at localization, it is highly complementary when combined with prior fault localization tools, even for ML-based approaches.

B. RQ2: Patch Generation Efficiency

In this section, we discuss the observed relationship between entropy and test-passing patches. We use entropy from InCoder, the most successful LLM in RQ1's fault localization. We measure the impact of entropy delta on patch generation efficiency with two methods: (1) measuring each successful (test-passing) patch's ranking as ranked by original TBAR and

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

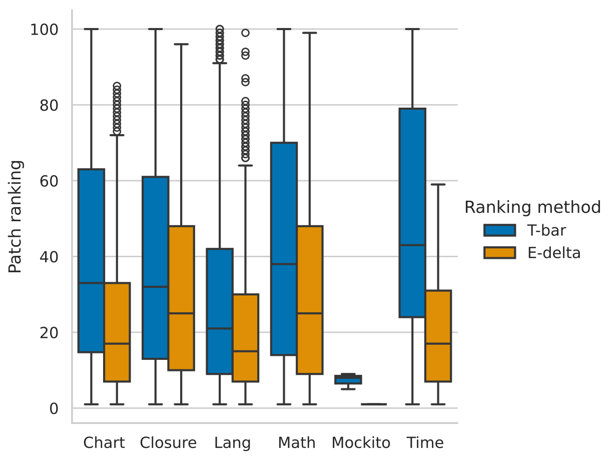


Fig. 4: Entropy delta and TBAR ranking (lower is better) of test-passing patches on 72 DEFECTS4J bugs.

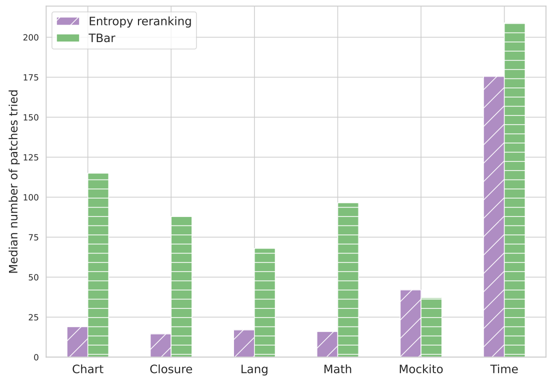


Fig. 5: Patches tested (median, lower is better) per project before successful patch using TBAR original ranking and entropy delta re-ranking of patches on 72 DEFECTS4J bugs.

entropy delta re-ranked TBAR, and (2) incorporating entropy delta into TBAR and measuring the total number of patches generated to pass all tests.

We first configured TBAR to generate only 100 patches per bug, assuming perfect fault localization. Recall that the dataset in question considers DEFECTS4J bugs that TBAR is known to be able to repair; for all 72 considered bugs, TBAR found at least one plausible patch that caused the program to pass all relevant tests (e.g., all tests written for project Chart) within the 100-patch bound. Finally, we calculated the entropy delta score for each patch, and the test-passing patches' original ranking according to TBAR. As seen in Table III, entropy delta improves 60 out of the 72 rankings as compared to TBAR's original ranking. On average, we observed a mean rank decrease of 24, meaning that using entropy delta to rank the generated TBAR patches can reduce a mean of 24 full test iterations (i.e., each potential patch must run through all test cases in the repository before knowing if it is a plausible patch). Liu et al. [25] compared 16 APR techniques and found that TBAR exhibits one of the highest number of patches generated and the highest rate of bug fixing across DEFECTS4J. We posit that entropy delta's efficiency improvement over TBAR significantly boosts template-based APR's overall utility.

Figure 4 compares the TBAR ranking and entropy delta ranking. Each bar represents the rank of test-passing patches compared to all generated patches per DEFECTS4J project. A lower rank signifies a more efficient repair process, as the repair process ends when a test-passing patch is found. As seen in Figure 4, TBAR's original ranking for test-passing patches is higher than entropy delta's ranking across all projects. Entropy delta shows a higher disparity on ranking between test passing and test failing patches (i.e., a lower median rank for all test-passing patches). In particular, patches from projects Chart and Time show the largest improvement from re-ranking patches with entropy delta. Successful patches in Chart and Time typically require multi-line edits, and with a wider range of templates to choose from, entropy delta can make a greater impact in reducing the number of patches tested.

We then configured TBAR to use entropy delta ranked patches directly, and measured the total number of patches required until a successful bug fix (i.e., passing all tests). Figure 5 shows the median number of patches tested per project before a successful patch using TBAR original ranking and entropy delta re-ranking. We observe that entropy delta re-ranking reduces the median number of patches tested across all projects except for Mockito. Mockito has only three single line bugs that TBAR can fix. With a smaller total number of patches to try on a single template (e.g., 11 total possible patches for Mockito-26), entropy delta re-ranking does not have as large of an impact on APR efficiency.

RQ2 Summary

We show that entropy can be used to rank patches before evaluating them via a test suite, improving efficiency for template-based technique TBAR by reducing a mean of 24 patches tested per bug. Entropy delta can both reduce the median number of patches tried before finding a fix, and consistently rank test-passing patches higher than test-failing patches, without any analysis of the test suite. Entropy delta is most useful for bugs that require multi-line patches.

C. RQ3: Patch correctness

In RQ2, we saw that entropy delta can improve the efficiency of patch generation by reducing number of patches tested. However, a test-passing patch is not necessarily correct. We therefore now apply entropy deltas to distinguish between correct and and overfitting (incorrect) patches.

1) Patch ranking: We evaluate entropy delta's ability to rank patches on a dataset of 1,290 patches collected by Ghanbari et al. [28]. For each bug, the data set includes some number of plausible (i.e., test passing) patches, where exactly one is correct, and the rest are incorrect. We attempt to isolate the true correct patch from the incorrect patches. We then rank each patch according to its entropy delta, querying the model for the entropy of the entire patch region before and after the replacement. Following the approach by Shibboleth [28], we

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

TABLE IV: Ranking results of 1290 plausible patches per DEFECTS4J project using ranking methods SBFL, Shibboleth, and entropy delta

Project	#Patches	#Correct	#Incorrect	Top-N	SBFL	Shibboleth	Entropy delta
Chart	201	19	182	Top-1	3	11	10
				Top-2	6	14	14
Closure	269	64	205	Top-1	19	27	48
				Top-2	38	47	58
Lang	220	35	185	Top-1	1	14	20
				Top-2	12	22	27
Math	541	67	474	Top-1	10	27	39
				Top-2	30	38	55
Mockito	2	1	1	Top-1	0	1	1
				Top-2	1	1	1
Time	57	11	46	Top-1	3	8	9
				Top-2	5	5	10
Total	1290	197	1093	Top-1	36	85	**127**
				Top-2	92	130	**165**

TABLE V: Classification scores of 2,147 plausible patches on DEFECTS4J projects using PATCH-SIM, Panther, and entropy delta

Score	PATCH-SIM	Panther	Entropy delta
Accuracy	0.388	0.730	0.735
Precision	0.245	0.760	0.900
+ Recall	0.711	0.757	0.760
- Recall	0.572	0.696	0.624
F1	0.377	0.750	0.824

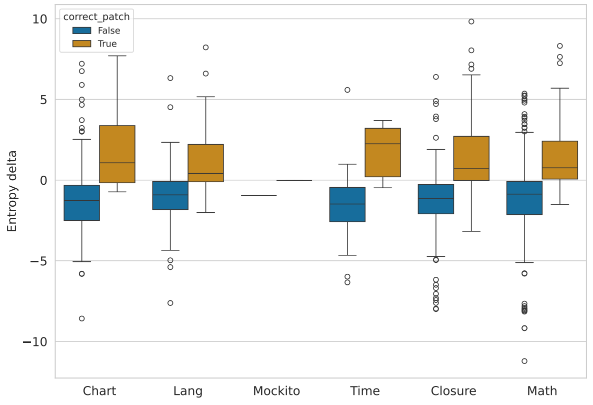


Fig. 6: Entropy delta across correct and incorrect patches on DEFECTS4J projects. A higher entropy delta signifies a less surprising patch, and a lower entropy (sometimes negative) entropy delta signifies a more surprising patch to the LLM.

use Ochiai's SBFL for patch ranking as a baseline: ranking patches by the SBFL suspiciousness of the modified location.

Table IV shows the Top-1 and Top-2 results of our approach on the 1,290 labeled patches. Table IV shows that entropy delta outperforms both SBFL and Shibboleth [28] on Top-2 across all projects; it delta outperforms Shibboleth on Top-1 across all projects but Chart (10 vs. 11 Top-1). Overall, entropy delta improves upon Shibboleth by 49% for Top-1, and 27% for Top-2.

Figure 6 illustrates the difference in entropy reduction between correct and plausible-but-incorrect patches in finer detail, via a clear difference in entropy delta between correct and incorrect patches. In particular, the correct patches for all projects have a median entropy delta value of above 0, while incorrect patches for all projects have a median entropy delta value of below 0. A correct patch tends to appear more natural to the LLM as compared to its original buggy line.

2) *Patch classification:* Table V shows our classification results on a labeled dataset of 2,147 plausible patches curated by Tian et al. [27] for classifying patches as correct or incorrect. Entropy delta improves upon the accuracy score of PATCH-SIM [5] and Panther [27], but only slightly improves +recall score over both PATCH-SIM and Panther. For -recall, entropy delta performs better than PATCH-SIM by 9%, but performs worse than Panther by 10%. Entropy delta slightly improves accuracy over Panther by 0.6%, and 89% over PATCH-SIM. Entropy delta improves precision over Panther by 18%, and PATCH-SIM by 267%. Finally, entropy delta performs better than both PATCH-SIM and Panther on F1 score, by 118% and 10% respectively. As compared to the state-of-the-art, entropy improves classification performance on true positives more than true negatives.

3) *Patches produced by an ML-based approach:* While rigorously constructed and validated, the datasets we consider so far are produced by APR techniques that pre-date modern neural- and LLM-based innovations. We therefore also consider patches produced by ITER [29], a state-of-the-art machine learning based APR technique. ITER's replication package includes 79 bugs with correct-overfitting patch pairings: the correct patches are determined to be correct via comparison to the developer fix, while the overfitting patches are plausible (test-passing) but incorrect. Entropy delta ranks the correct patch higher than the plausible but incorrect patch in 68 out of 79 correct-plausible patch pairings.

Overall, our analysis suggests that correct patches tend to lower entropy (i.e., increase naturalness) more than incorrect patches. Specifically, entropy delta ranks 49% more correct

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

patches in the Top-1 than the state-of-the-art patch ranker Shibboleth, and entropy delta can classify correct patches with an 18% higher precision than the state-of-the-art patch classifier Panther. Entropy delta ranks the correct patch higher than a corresponding overfitting patch for 86% of patch pairs produced by the ML-based APR tool ITER. These findings indicate that entropy deltas can be a valuable heuristic for distinguishing between correct and incorrect patches, for patches produced by a variety of techniques.

RQ3 Summary

The entropy delta from an LLM distinguishes between correct and plausible (test-passing but incorrect) patches with higher precision and accuracy than state-of-the-art patch disambiguation tools. Entropy delta can also select the correct patch from two plausible patches 68 out of 79 times when repairing with an ML-based APR tool.

VI. Discussion

We provide a discussion of threats, limitations, and potential end-to-end implications of our results.

A. Threats

External validity. A threat to our study's external validity is the potential selection bias of our LLMs. We chose these three based on their infill ability and built-in bidirectional attention mechanism; this set is representative, with a range of trainable parameters. Much larger LLMs ($>$ 20 billion parameters) might be better able to reason over faulty code and patches, but require far greater computation power and time for entropy calculation. Another threat to external validity is our use of Defects4J data. We chose Defects4J for our target bugs for fault localization and patch disambiguation due to the data available and the aim to compare against related work in APR. Data leakage of Defects4J as training data for our selected LLMs is possible. We mitigate this risk by (1) primarily using entropy in combination with prior APR techniques instead of direct LLM prompting for patch generation, and (2) applying entropy delta on untested or plausible patches that are not necessarily readily available online (i.e., recently generated and not used as an official patch for bug fixing).

Internal validity. A threat to the internal validity of our results is the manual labeling of plausible patches. We used manually labeled data released by prior works [27], [28], in which the authors followed clear and reproducible decision criteria. Although mistakes could still be made on which plausible patches are correct or incorrect, we use the same labeling for all prior tools studied as well as entropy to create a standardized baseline on patch classification.

Construct validity. One risk to construct validity is the measurements we chose for our empirical evaluation. We used Top-N as a ranking measurement for both bugs and patches, following prior work in APR and fault localization. To overcome limitations of Top-N and strengthen claims of generalizability, we also use multiple patch classification measurements (accuracy, precision, recall, and F1) on a separate set of labeled patch data.

B. End-to-end implications

In this work, we sought to isolate the potential utility of LLM entropy in augmenting the primary components of the program repair problem: fault localization, patch generation, and patch evaluation. Our study design enabled a controlled assessment of these effects. However, this design leaves the overall end-to-end implications of entropy for repair largely open. We focused on an extant APR system, TBar, that includes no learned components, specifically to disambiguate the potential effects of entropy in patch generation. That said, the controlled setting of RQ2 offers a glimpse into the potential overall effects of entropy, even incorporated narrowly (Section V-B). We note that within the first 100 potential patches generated and then reevaluated in order of entropy delta on the considered Defects4J benchmarks, TBar: (a) (correctly) repairs two additional defects within the given timeout (where without entropy, TBar produced no patch at all),[5] and (b) correctly (in a developer-patch-equivalent way) repairs two other defects with the first plausible (test-passing) patch produced, where vanilla TBar's first plausible patch is overfitting/incorrect. This is in addition to the 46 other defects (of the 72 total considered) that both vanilla- and entropy-enhanced-TBar repair correctly with the first plausible patch. In these cases, as discussed, entropy-enhanced TBar finds a correct, developer-equivalent patch significantly faster. We view these initial results as motivating the design of mechanisms to allow statistical and programmatic analyses to collaborate most effectively.

VII. Related Work

We discuss in the following sections the most recent advances in LLM for code, fault localization, and patch ranking.

A. LLM for code

Language models have been used for code generation, bug detection, and patch generation. Recent language models finetune on code as training data and can perform code completion [31], [40], and generate code based on natural language [41] with impressive results. Large Language Models (LLMs), such as CodeLlama-2 [35], have improved performance on these tasks by using more trainable parameters and training data. Ray et al. [15] study the relationship between bugginess and LLM-entropy, showing that n-gram models trained over large code corpora code find buggy statements more surprising.

In closely-related recent work, Kolak et al. [42] investigate patch naturalness, using large language models ranging from 160M to 12B parameters to produce and compare LLM-generated- with developer-written-patches for single-line bugs. They find that larger models are both more effective at generating test-passing patches, and their patches are more similar to

[5]Results from ref [25], where experimental setup/goals match ours.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

the human-written gold standard. Xia et al. [19] directly apply LLMs for APR, with several implications, e.g., LLMs can suggest multi-line fixes more accurately than the previous state-of-the-art. One particularly salient takeaway that substantiates some of our conclusions is that while LLMs can often perform localization and generation in one shot, it is significantly more cost-effective to use traditional localization first/instead. Both Kolak et al. [42] and Xia et al. [19] focus on patch *generation* specifically. By contrast, our empirical evaluation investigates all three stages of APR. To that end, we hold patch generation constant by primarily using the template-based repair tool TBAR, which does not incorporate a learned component.

B. Fault localization

Prior fault localization tools use test output information, code semantics, and naturalness of code to achieve a high degree of confidence on bug detection. Spectrum-based Fault Localization (SBFL) [24], [43] uses a ratio of passed and failed tests covering each line of code to calculate its suspiciousness score, in which a higher suspiciousness signifies a higher probability of being faulty. Recent advances in deep learning created a spur of research on using graph neural networks (GNNs) [44] for fault localization. GRACE [45], DeepFL [46], and DEAR [33] encode the code AST and test coverage as graph representations before training deep learning models for fault localization. TransferFL [23] combined semantic features of code and the transferred knowledge from open-source code data to improve the accuracy of prior deep learning fault localization tools. LLMAO [14] finetuned a lightweight bidirectional layer on top of code-tuned LLMs to show that LLMs can detect both bugs and security vulnerabilities without the use of test cases. Our work builds on top of the top-performing prior fault localization tools and shows that entropy can be used as a lightweight re-ranking tool that improves fault localization scores without a dependency on test cases.

C. Patch correctness

Similarly to prior fault localization tools, prior patch disambiguation tools leverage test output and code information (both syntax and semantics) for ranking or classifying patches. Qi et al. [47] analyzed the reported bugs of three generate-and-validate APR tools, finding that producing correct results on a validation test suite is not enough to ensure patch correctness. Smith et al. [26] performed a controlled experiment on patch quality, and, borrowing the idea of training and test sets from machine learning, dubbed the quality problem produced by "training" (generating a patch), and "testing" (validation) on the same test suite, "overfitting".

Subsequent work has aimed to counteract the overfitting problem in several ways. Ye et al. [48] proposed ODS (Overfitting Detection System), that statically classifies overfitting patches. Xiong et al. [5] generated both execution traces of patched programs and new tests to assess the patch correctness. Ghanbari et al. [28] propose Shibboleth, which uses both the syntactic and semantic similarity between original code and proposed patch, and code coverage of passing tests, to rank patches. Tian et al. [27] proposed Panther, a predictor with transformer-based learned embeddings for patch classification. We use the datasets from Panther and Shibboleth for the majority of our patch correctness experiments.

The most relevant work to our study of patch correctness is Yang et al. [20], who showed that state-of-the-art learning-based techniques suffered from the dataset overfitting problem, and that naturalness-based techniques outperformed traditional static techniques, like Patch-Sim [5]. By contrast, we use datasets of 2,147 plausible patches collected most recently in 2023, lowering the risk of LLM training data leakage. We perform an empirical study comparing entropy against the most recent state-of-the-art patch disambiguation techniques Panther [27] and Shibboleth [28], on top of Patch-Sim [5].

Khanfir et al. [18] developed a mechanism to compute entropy from CodeBERT, contributing to the evidence [15] that naturalness statistically differs between buggy and correct code. We build on their findings by combining and comparing LLM-based entropy with and against prior approaches, showing that a combination is most effective. Motivated by Liu et al. [25], our work is the first to use LLM entropy on plausible patches before undergoing testing to achieve more efficient APR on prior template-based techniques. Finally, we introduce a new naturalness measurement for patches, entropy delta, which achieves state-of-the-art results for plausible patch disambiguation without depending on the test-suites of buggy programs, which lowers the risk of dataset overfitting.

VIII. Conclusion

In this work, we propose the use of "unnaturalness" of code for automated program repair through the measurement of entropy generated by code-tuned LLMs. We also introduce entropy delta, which measures the difference in entropy between a proposed code insert (i.e., a patch) and the original code. Using three LLMs and three prior fault localization tools, we show that entropy can improve Top-5, 3, and 1 scores after re-ranking the first 6 potential bug localization. We use entropy delta on untested patches to save an average of 24 test runs per bug for the template-based APR technique TBAR. We show that entropy delta can improve upon state-of-the-art patch ranking by 49% for Top-1, can classify plausible patches with a 18% higher precision, and is still effective for patches produced by modern ML-based tools. Our results indicate that LLMs can be a powerful addition to state-of-the-art APR tools without the dependency on tests, and the usage of LLM code generation. The reduction in both test suites and LLM code generation results in the reduction in model overfitting and training data leakage.

Data and Artifact. The archival replication package can be found at https://zenodo.org/records/10851256, and https://github.com/squaresLab/entropy-apr-replication. The extended version of TBar is available at https://github.com/squaresLab/TBar.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

References

[1] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[2] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[3] D. Wu and J. M. Mendel, "Patch learning," *IEEE Transactions on Fuzzy Systems*, vol. 28, no. 9, pp. 1996–2008, 2019.

[4] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[5] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 789–799.

[6] M. Kim, Y. Kim, K. Kim, and E. Lee, "Multi-objective optimization-based bug-fixing template mining for automated program repair," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–5.

[7] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: scalable multiline program patch synthesis via symbolic analysis," in *International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 691–701.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[9] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[10] S. Black, S. Biderman, E. Hallahan, Q. Anthony, L. Gao, L. Golding, H. He, C. Leahy, K. McDonell, J. Phang *et al.*, "Gpt-neox-20b: An open-source autoregressive language model," *arXiv preprint arXiv:2204.06745*, 2022.

[11] C. Koutcheme, S. Sarsa, J. Leinonen, A. Hellas, and P. Denny, "Automated program repair using generative models for code infilling," in *International Conference on Artificial Intelligence in Education*. Springer, 2023, pp. 798–803.

[12] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "Inferfix: End-to-end program repair with llms," *arXiv preprint arXiv:2303.07263*, 2023.

[13] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*, 2023.

[14] A. Z. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, "Large language models for test-free fault localization," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[15] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the" naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 428–439.

[16] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[17] P. F. Brown, V. J. Della Pietra, P. V. Desouza, J. C. Lai, and R. L. Mercer, "Class-based n-gram models of natural language," *Computational linguistics*, vol. 18, no. 4, pp. 467–480, 1992.

[18] A. Khanfir, M. Jimenez, M. Papadakis, and Y. Le Traon, "Codebert-nt: code naturalness via codebert," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2022, pp. 936–947.

[19] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1482–1494.

[20] J. Yang, Y. Wang, Y. Lou, M. Wen, and L. Zhang, "A large-scale empirical review of patch correctness checking approaches," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1203–1215.

[21] S. Balloccu, P. Schmidtová, M. Lango, and O. Dušek, "Leak, cheat, repeat: Data contamination and evaluation malpractices in closed-source llms," *arXiv preprint arXiv:2402.03927*, 2024.

[22] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, 2006, pp. 39–46.

[23] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, "Improving fault localization and program repair with deep semantic features and transferred knowledge," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1169–1180.

[24] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing*. IEEE, 2006, pp. 39–46.

[25] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 615–627.

[26] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 532–543. [Online]. Available: https://doi.org/10.1145/2786805.2786825

[27] H. Tian, K. Liu, Y. Li, A. K. Kaboré, A. Koyuncu, A. Habib, L. Li, J. Wen, J. Klein, and T. F. Bissyandé, "The best of both worlds: Combining learned embeddings with engineered features for accurate prediction of correct patches," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–34, 2023.

[28] A. Ghanbari and A. Marcus, "Patch correctness assessment in automated program repair based on the impact of patches on production and test code," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 654–665.

[29] H. Ye and M. Monperrus, "Iter: Iterative neural repair for multi-location patches," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[30] 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*, 2014, pp. 437–440.

[31] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," 2022. [Online]. Available: https://arxiv.org/abs/2204.05999

[32] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 165–176.

[33] Y. Li, S. Wang, and T. N. Nguyen, "Dear: A novel deep learning-based approach for automated program repair," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 511–523.

[34] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, "Starcoder: may the source be with you!" *arXiv preprint arXiv:2305.06161*, 2023.

[35] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale *et al.*, "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[36] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[37] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[38] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[39] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 191–201.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.

[40] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, and S. Roy, “Program synthesis using natural language,” in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 345–356.

[41] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” in *Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation*, 2014, pp. 419–428.

[42] S. D. Kolak, R. Martins, C. Le Goues, and V. J. Hellendoorn, “Patch generation with language models: Feasibility and scaling behavior,” in *Deep Learning for Code Workshop*, 2022. [Online]. Available: https://openreview.net/forum?id=rHlzJh_b1-5

[43] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, “On the accuracy of spectrum-based fault localization,” in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[44] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, “The graph neural network model,” *IEEE transactions on neural networks*, vol. 20, no. 1, pp. 61–80, 2008.

[45] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, “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*, 2021, pp. 664–676.

[46] 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*, 2019, pp. 169–180.

[47] Z. Qi, F. Long, S. Achour, and M. Rinard, “An analysis of patch plausibility and correctness for generate-and-validate patch generation systems,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.

[48] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, “Automated classification of overfitting patches with statically extracted code features,” *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2920–2938, 2021.

Authorized licensed use limited to: CITY UNIV OF HONG KONG. Downloaded on September 03,2025 at 08:09:27 UTC from IEEE Xplore. Restrictions apply.