Challenging Bug Prediction and Repair Models with Synthetic Bugs

Ali Reza Ibrahimzada[1], Yang Chen[1], Ryan Rong[2], Reyhaneh Jabbarvand[1]
[1]*University of Illinois Urbana-Champaign*, Urbana, IL, USA [2]*Stanford University*, Stanford, CA, USA
{alirezai,yangc9,reyhaneh}@illinois.edu {ryanrong}@stanford.edu

***Abstract*—Bugs are essential in software engineering; many research studies in the past decades have been proposed to detect, localize, and repair bugs in software systems. Effectiveness evaluation of such techniques requires *complex bugs*, i.e., those that are *hard to detect through testing* and *hard to repair through debugging*. From the classic software engineering point of view, a hard-to-repair bug differs from the correct code in multiple locations, making it hard to localize and repair. Hard-to-detect bugs, on the other hand, manifest themselves under specific test inputs and reachability conditions. These two objectives, i.e., generating hard-to-detect and hard-to-repair bugs, are mostly aligned; a bug generation technique can change multiple statements to be covered only under a specific set of inputs. However, these two objectives conflict in the learning-based techniques: A bug should have a similar code representation to the correct code in the training data to challenge a bug prediction model to distinguish them. The hard-to-repair bug definition remains the same but with a caveat: the more a bug differs from the original code (at multiple locations), the more distant their representations are and easier to detect. This demands new techniques to generate bugs to complement existing bug datasets to challenge learning-based bug prediction and repair techniques.**

We propose BUGFARM to transform arbitrary code into multiple hard-to-detect and hard-to-repair bugs. BUGFARM mutates code in multiple locations (hard-to-repair) but leverages attention analysis to only change the least attended locations by the underlying model (hard-to-detect). Our comprehensive evaluation of $435k+$ bugs from over $1.9M$ mutants generated by BUGFARM and two alternative approaches demonstrates our superiority in generating bugs that are hard to detect by learning-based bug prediction approaches (up to 40.53% higher False Negative Rate and 10.76%, 5.2%, 28.93%, and 20.53% lower Accuracy, Precision, Recall, and F1 score) and hard to repair by state-of-the-art learning-based program repair technique (28% repair success rate compared to 36% and 49% of LEAM and μBERT bugs). BUGFARM is efficient, i.e., it takes nine seconds to mutate a code with no training overhead.

***Index Terms*—Bug Generation, Bug Prediction, Interpretation**

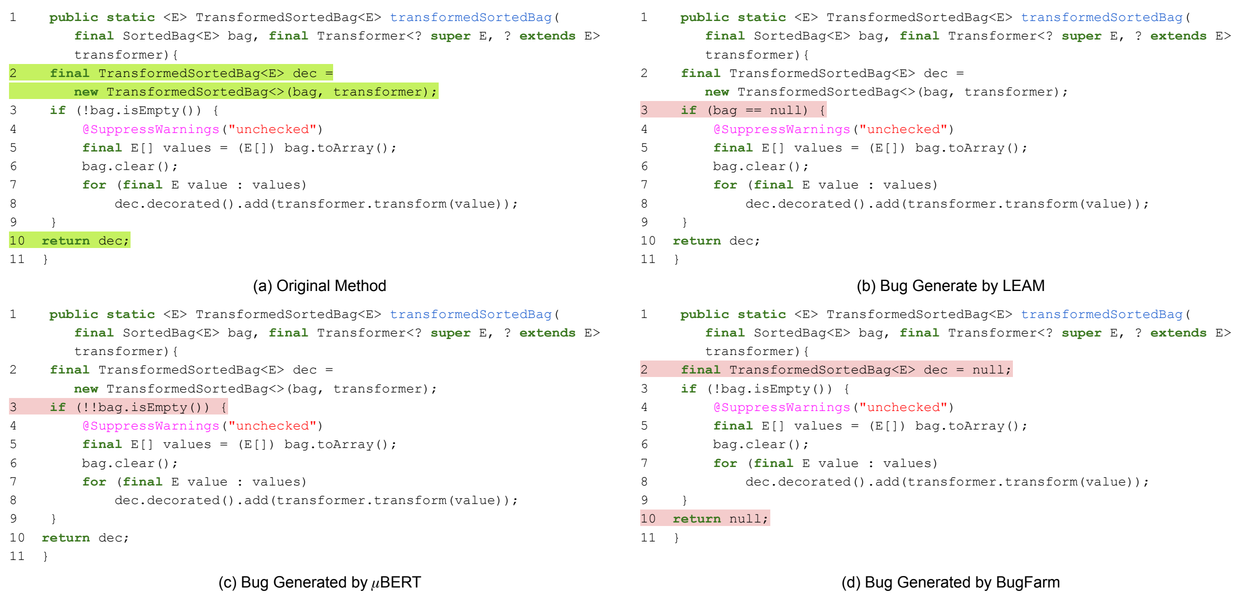
I. INTRODUCTION

Machine learning is the new automation technology, and software engineering is no exception: State-of-the-art software analysis techniques either fine-tune pre-trained models or prompt Large Language Models (LLMs) to automate code-related tasks [35], [10], [20], [30]. This demands the existence of high-quality datasets to assess their actual effectiveness. Concerning bug-related tasks, such datasets should include a diverse set of complex bugs, i.e., those that are hard to detect to challenge bug prediction techniques and hard to repair for debugging approaches.

From the classic software engineering perspective, hard-to-detect bugs often exist at locations reachable only under a particular combination of test inputs or edge cases [31], [5]. *However, a similar definition does not stand concerning learning-based bug prediction techniques.* That is because they do not care what inputs trigger the bug in the given code. In contrast, they look for bug patterns in their training/fine-tuning data [51] or to check if the code representation is closer to buggy or correct examples they have seen [21]. As a result, existing real-world bug datasets such as Defects4J [24], BUGSWARM [45], BugsInPy [8], RegMiner [42], and ManySStuBs4J [18], while *necessary* to assess their effectiveness in dealing with real-world bugs, *are not enough* to assess their true effectiveness and rule out data contamination [40]: To challenge the learning-based bug prediction techniques, one should inject unseen bug patterns such that the code representation of the generated bug and correct code are similar.

Hard-to-repair bugs usually involve multiple statements to challenge debugging techniques, localizing [29] and fixing automatically. A similar definition is held in learning-based software engineering, but with slight consideration: Modifying a code in multiple locations can change the representation of buggy code, making it easy for bug prediction models to detect it. These two conflicting objectives make the problem of complex bug generation for learning-based techniques challenging. Existing automated mutant generation techniques [44], [25], [22], by default, change a few locations in the code, making their bugs easy to repair and detect. Some can be configured to modify multiple lines randomly [25] or follow a set of heuristics [44]. However, this may result in mutants with a different code representation than the correct code, making them easy to detect.

To advance automated bug generation concerning the evaluation of learning-based bug-related tasks, we propose BUGFARM. For a given code, BUGFARM prompts an LLM to mutate multiple statements. Given that LLMs are potentially creative in generative tasks, leveraging them helps avoid overfitting to a limited number of mutation operators and bug patterns. To ensure that changing many locations does not drastically impact the code representation of mutants, BUGFARM analyzes the attention of the underlying model and instructs LLMs only to change those the model attends least to. The generated code has a similar representation to the original one. Still, it differs within multiple locations, making them hard to detect by bug prediction approaches and hard to repair by



```java
public static <E> TransformedSortedBag<E> transformedSortedBag(
    final SortedBag<E> bag, final Transformer<? super E, ? extends E>
    transformer){
  final TransformedSortedBag<E> dec =
    new TransformedSortedBag<>(bag, transformer);
  if (!bag.isEmpty()) {
     @SuppressWarnings("unchecked")
     final E[] values = (E[]) bag.toArray();
     bag.clear();
     for (final E value : values)
         dec.decorated().add(transformer.transform(value));
  }
 return dec;
 }
```

(a) Original Method

```java
public static <E> TransformedSortedBag<E> transformedSortedBag(
    final SortedBag<E> bag, final Transformer<? super E, ? extends E>
    transformer){
  final TransformedSortedBag<E> dec =
    new TransformedSortedBag<>(bag, transformer);
  if (bag == null) {
     @SuppressWarnings("unchecked")
     final E[] values = (E[]) bag.toArray();
     bag.clear();
     for (final E value : values)
         dec.decorated().add(transformer.transform(value));
  }
 return dec;
 }
```

(b) Bug Generate by LEAM

```java
public static <E> TransformedSortedBag<E> transformedSortedBag(
    final SortedBag<E> bag, final Transformer<? super E, ? extends E>
    transformer){
  final TransformedSortedBag<E> dec =
    new TransformedSortedBag<>(bag, transformer);
  if (!!bag.isEmpty()) {
     @SuppressWarnings("unchecked")
     final E[] values = (E[]) bag.toArray();
     bag.clear();
     for (final E value : values)
         dec.decorated().add(transformer.transform(value));
  }
 return dec;
 }
```

(c) Bug Generated by μBERT

```java
public static <E> TransformedSortedBag<E> transformedSortedBag(
    final SortedBag<E> bag, final Transformer<? super E, ? extends E>
    transformer){
  final TransformedSortedBag<E> dec = null;
  if (!bag.isEmpty()) {
     @SuppressWarnings("unchecked")
     final E[] values = (E[]) bag.toArray();
     bag.clear();
     for (final E value : values)
         dec.decorated().add(transformer.transform(value));
  }
 return null;
 }
```

(d) Bug Generated by BugFarm

Fig. 1: An example showing the bugs generated for the original code (a) using LEAM (b), μBERT (c), and BUGFARM (d)

debugging techniques. BUGFARM is language-agnostic and can generate bugs for any programming language. Like alternative approaches, the BUGFARM creates unconfirmed bugs and requires a validation process to confirm them as bugs. Our notable contributions are:

- **A new perspective into synthetic bug generation:** We propose a novel technique for bug generation concerning the evaluation of the learning-based bug-related tasks. We do not claim our bugs challenge classic bug detection (testing) techniques, as we propose a new definition for hard-to-detect bugs. Our goal is to demonstrate how ignoring properties of learning-based techniques in mutant/bug generation results in subpar assessment of them. The implementation of BUGFARM and all the generated bugs are publicly available [27].
- **Empirical evaluation:** We evaluated BUGFARM and two most recent learning-based mutant generation approaches, LEAM [44] and μBERT [25]. Our evaluation of $435k+$ bugs (from over $1.9M$ mutants) generated for 15 Java projects confirm that compared to others, BUGFARM bugs are hard-to-detect (up to 40.53% higher False Negative Rate and 10.76%, 5.2%, 28.93%, and 20.53% lower Accuracy, Precision, Recall, and F1 score) and hard-to-repair (28% repair success rate compared to 36% and 49% of LEAM and μBERT bugs) by learning-based techniques. A large-scale human study with 97 participants and 500 sampled survived mutants from these techniques shows that BUGFARM mutants are harder to decide buggy or equivalent. Also, generating effective bugs is more efficient using BUGFARM.

II. Illustrative Example

To illustrate the limitations of prior work and present the key ideas behind BUGFARM, we use the code snippets in Figure 1. There are several ways to inject bugs into original code by adding, deleting, or modifying the 11 statements in its body (some statements are split into two lines for better presentation). However, the highlighted lines in green show the two least attended statements (details in algorithm 1).

Figures 1b–1d show the bugs generated by three mutant generation tools, BUGFARM, LEAM [44], and μBERT [25] (bugs were confirmed through test execution). LEAM considers code as a sequence of AST nodes and learns to apply grammar rules to select and modify the code for bug generation. μBERT selects code tokens corresponding to AST nodes, replaces them with a special token `<mask>`, and asks CodeBERT to replace them with new tokens for bug generation. BUGFARM identifies the least attended statements and prompts an LLM to perform bug-inducing transformations only on those lines. LEAM and μBERT generate 19 and 68 mutants in total, out of which only 6 and 1 are confirmed bugs, respectively. BUGFARM generates the bug in Figure 1d. The lines changed to introduce a bug are highlighted in red.

BUGFARM bug is notable from various perspectives. First, it involves multiple statements (hard to localize and repair). Second, both modified statements are among the least attended statements, making it hard for the model to distinguish the bug from the original code (hard to detect). LEAM and μBERT bugs modify only one line that is not among the least attended statements. As a result, when running against our studied bug prediction model (§V-C) and program repair model (§V-D), they were easily detected and repaired. In contrast, the same models failed to detect and repair BUGFARM bug.

III. Approach Overview

Figure 2 provides an overview of BUGFARM framework consisting of *three* major components: (1) *Method Extractor*, (2) *Attention Analyzer*, and (3) *Bug Generator*. BUGFARM takes a project and a pre-trained code-language model as inputs and extracts the methods through a lightweight static analysis. For each method, it identifies the change location candidates by analyzing the model's attention to individual code tokens. To

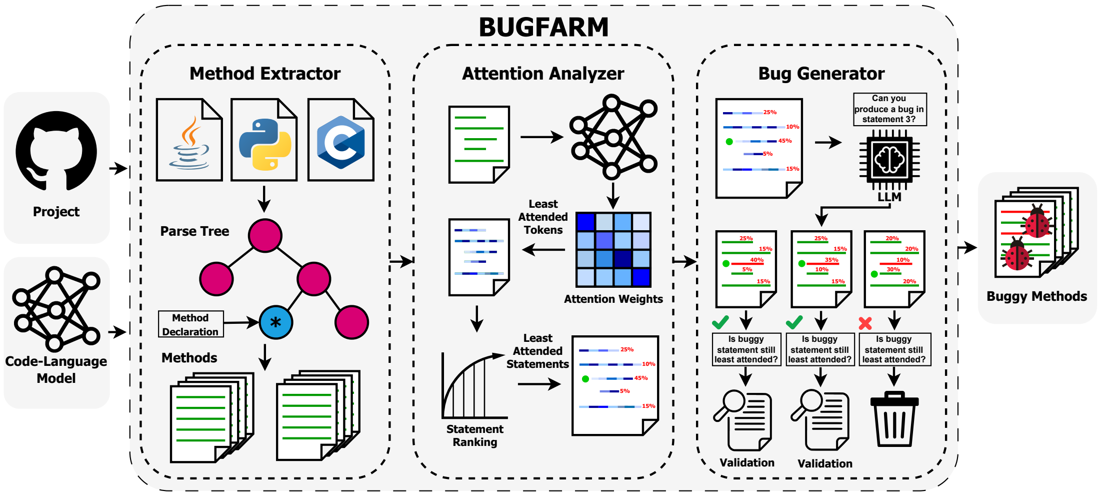


Fig. 2: Overview of BugFarm

generate mutants for each method, BugFarm crafts a prompt to an LLM, including the original code and additional contexts reflecting candidate locations for mutations.

The *Method Extractor* component takes the input project and builds its corresponding parse tree to extract all the methods in the source files. These methods will be passed as an input to *Attention Analyzer* to identify the candidate locations for the mutation (§IV-A).

Recent state-of-the-art learning-based software engineering models are all transformer-based, which rely on attention for neural code representation. So, to identify the candidate locations to inject unnoticeable bugs, *Attention Analyzer* extracts the corresponding attention of the model to code tokens and identifies those with the lower contribution in the code representation, i.e., those with less attention weight values. Bug locations in BugFarm are at the statement level; thereby, BugFarm ranks the input method's statements based on the $\#T_{LA}/\#T$ values and chooses the ones with the lowest value. Here, T_{LA} is the number of least attended tokens, and T denotes the total number of tokens in the statement (§IV-B).

Finally, *Bug Generator* component takes the list of location candidates and creates a prompt consisting of natural language instruction, location candidates, and the original code. For the generated mutants, BugFarm computes the extent to which the changes impact the model's attention, and selects mutants with negligible impact on the code representation. It also discards the syntactically incorrect bugs for the sake of quality (§IV-C).

IV. BugFarm

In this section, we first discuss how BugFarm parses the subjects and extracts methods as inputs for the *Attention Analyzer* component. Next, we will provide a background on the attention analysis required to understand the subsequent components. Finally, we answer two main questions, namely (1) How does BugFarm decide *where* to inject bugs?, and (2) *how* BugFarm generates a set of bugs?

A. Method Extractor

BugFarm can take a single method or a repository as input. In the latter case, it first needs to extract the implemented methods in the project for further mutation. To that end, *Method Extractor* component leverages a parser depending on the programming language used in the input project, builds program parse trees, and extracts a list of methods and constructors. Next, it collects the signature and body of the method (excluding docstrings) and passes them to the *Attention Analyzer* component.

B. Attention Analyzer

State-of-the-art code-language models are based on the Transformer architecture [47]. To produce contextualized vector representation of a sequence of tokens, Transformers rely on *Multi-Head Self-Attention*. For a method that consists of n tokens $\overrightarrow{Tkn} = \{m_0, \ldots, m_{n-1}\}$, a Transformer model with $\mathcal{L}$ layers takes $\overrightarrow{Tkn}$ as input and produces $\mathbf{H}^{\ell} = [\mathbf{h}_0^{\ell}, ..., \mathbf{h}_{n-1}^{\ell}]$. Here, $\mathbf{H}^{\ell}$ corresponds to the hidden vector representations in layer $\ell \in \{1, 2, ..., \mathcal{L}\}$. In each Transformer layer ℓ, multiple self-attention heads are used to aggregate the previous layer's outputs. Consequently, for each token m_i, the self-attention assigns a set of attention weights concerning all other tokens in the input sequence, i.e., $Attention(m_i) = \{\alpha_{i0}, \ldots, \alpha_{in-1}\}$, where α_{ij} indicates the relative attention of m_i to m_j.

Attention weights reflect the importance of each token in the final code representation. Hence, BugFarm analyze attention weights to identify tokens (and subsequently, statements) with the lowest attention weights. These statements are where BugFarm can change during bug-inducing transformation without impacting the overall representation of the method. Algorithm 1 explains our approach for attention analysis, which takes the original method, a threshold value k, and a transformer-based model as inputs and pinpoints the $k\%$ of the least attended statements LAS as outputs. To that end, it first extracts the list of least attended tokens in the method LAT (Lines 1-10) and uses them to pinpoint the least attended statements LAS (Lines 11-20).

The algorithm first identifies the tokens $\overrightarrow{Tkn}$ and statements $\overrightarrow{Smt}$ in the given method and initializes the LAT and LAS variables to be empty (Lines 1-3). Next, it queries the model M to extract the self-attention values (Line 4). For a model M with L layers and H attention heads per layer, the attention values will be averaged across heads and layers, resulting in an $n \times n$ matrix, where n is the number of tokens. For each token m_i in the $method$, the algorithm further averages the attention weight relative to other tokens (averaging the values per column in the self-attention matrix) to compute a single attention weight value for each token m_i in the method (Line 5). Given that we are interested in the least attended tokens in the code, Algorithm 1 sorts the attention weight vector, $\overrightarrow{TknAttnW}$, populates their corresponding indices in $\overrightarrow{SortedTknInd}$ (Line 6), and identifies the top $k\%$ of least attended tokens, LAT (Lines 7-10).

Algorithm 1: Attention Analyzer

```
Inputs: Method $method$, Threshold $k$, Transformer-based model $M$
Output: Least attended statements $LAS$
1  $\overrightarrow{Tkn} \leftarrow getTokens(method)$;
2  $\overrightarrow{Smt} \leftarrow getStatements(method)$;
3  $LAT, LAS \leftarrow \emptyset$;
4  $SelfAttnW \leftarrow getSelfAttnW(M, \overrightarrow{Tkn})$;
5  $\overrightarrow{TknAttnW} \leftarrow getTknAttnW(SelfAttnW)$;
6  $\overrightarrow{SortedTknInd} \leftarrow getSortedTknIndices(TknAttnW)$;
7  $i \leftarrow 0$;
8  while $i < \lceil (k/100) * SortedTknInd.length \rceil$ do
9      $LAT \leftarrow LAT \cup Tkn[SortedTknInd[i]]$;
10     $i \leftarrow i + 1$;
11 $SmtScore \leftarrow \emptyset$;
12 foreach $s_i \in Smt$ do
13     $score \leftarrow |s_i \cap LAT| / s_i.length$;
14     $SmtScore \leftarrow SmtScore \cup \langle s_i, score \rangle$;
15 $SortedSmtInd \leftarrow getSortedSmtIndices(SmtScore)$;
16 $i \leftarrow 0$;
17 while $i < \lceil (k/100) * SortedSmtInd.length \rceil$ do
18     $LAS \leftarrow LAS \cup Smt[SortedSmtInd[i]]$;
19     $i \leftarrow i + 1$;
20 return $LAS$
```

With the least attended tokens extracted, the algorithm can identify the least attended statements, LAS. To that end, it weighs each statement by a score (Lines 11-14), which is the ratio of the number of least attended tokens in that statement, normalized by its length. The key idea here is that a statement with the highest overlap between least attended tokens should achieve a lower score and, thus, be considered the least attended statement. Without normalizing the statement length, longer statements will be penalized, i.e., BUGFARM never selects them to mutate. Finally, the statements are sorted in ascending order based on their scores, and the least $k\%$ attended statements (we take the ceiling in case $k\%$ of total statements is less than one) will be returned as LAS (Lines 15-20).

C. Bug Generator

Bug generation involves modifying, adding, or deleting code segments from an original method to change the expected behavior of the program. The *Bug Generator* module of BUGFARM takes a method and its corresponding set of LAS identified in the previous step as inputs and crafts LLM prompts to generate N buggy versions for the method as outputs. Our intuition for making BUGFARM configurable is that our bugs will likely be used as training/fine-tuning data for bug-related tasks. So, generating multiple buggy versions of a single method would be helpful for a model to distinguish between buggy and non-buggy code more easily. The total number of bugs, however, also depends on the size of the method and threshold value k. For example, $k = 10\%$ for methods with less than 10 statements returns one statement as LAS, and changing that statement in N unique ways may be infeasible.

Algorithm 2: Bug Generator

```
Inputs: Method $method$, Least attended statements $LAS$, Number
        of bugs $N$, Transformer-based model $M$
Output: Buggy methods $Bugs$
1  $Bugs \leftarrow \emptyset$;
2  $LASInd \leftarrow getLASIndices(method, LAS)$;
3  $method \leftarrow addIndices(method)$;
4  $Prompt \leftarrow$ "Inject \$$N$ bugs in the following method by changing
     only the statements at locations \$$LASInd$: \$$method$";
5  $Responses \leftarrow queryLLM(Prompt)$;
6  foreach $Response \in Responses$ do
7      if $!isParseable(Response)$ then
8          $continue$;
9      $newLAS \leftarrow getLAS(Response, M)$;
10     $check \leftarrow true$;
11     foreach $stmt \in getDiff(Response, method)$ do
12         if $stmt \notin newLAS$ then
13             $check \leftarrow false$;
14             $break$;
15     if $check$ then
16         $Bugs \leftarrow Bugs \cup Response$;
```

BUGFARM's prompts consist of three parts. The first part is the natural language instruction asking LLM to generate the bugs. The second part provides contextual information about where to inject the bug, i.e., only to consider the least attended statements, LAS, for making bug-inducing changes. Finally, we include the entire method, including both signature and the method body, in the prompt. Such prompts are LLM-agnostic, i.e., they can be used with various existing LLMs, such as LLaMa [32], PaLM [11], Copilot [17], Alpaca [43], and ChatGPT [2]. The only consideration is to check if the prompt's size matches the LLM context window. After the LLM's response, *Bug Generator* component validates (1) if they are syntactically correct and (2) if the changes do not impact the attention of the model. The responses that do not pass these two checks will be discarded.

Algorithm 2 demonstrates BUGFARM's bug generation and validation approach. It starts by initializing the output variable $Bugs$ (Line 1) and getting the indices of LASs in $method$ (Line 2). Next, it adds an index to each statement in $method$ (Line 3). This will help us to refer to LASs in the prompt by number instead of including the whole statements, resulting in a reduction of the prompt size. This is specifically important for longer methods or statements, as the context window of LLMs is limited [33], [50]. Next, we will craft the prompt with the required context, i.e., indexed method and bug injection location, and send the prompt to an LLM (Lines 4-5) [1]. Finally, once the LLM responds, we check if the generated bugs are syntactically correct (Lines 7-8) and if the changed statements are among the least attended statements by the model (Lines 9-14) to add them to the acceptable set of bugs (Lines 15-16).

V. Evaluation

To evaluate the effectiveness of BugFarm, we compare it with two state-of-the-art alternative approaches, namely LEAM and μBERT, investigating the following research questions [2]:

RQ1: Characteristics of the Generated Bugs. To what extent can BugFarm and alternative techniques successfully inject bugs into arbitrary code? What are the characteristics of the generated bugs by each approach?

RQ2: Effectiveness in Generating Hard-to-Detect Bugs. How well do *learning-based* bug prediction models perform at detecting BugFarm bugs compared to other techniques?

RQ3: Effectiveness in Generating Hard-to-Repair Bugs. To what extent do *learning-based* Automated Program Repair (APR) techniques repair BugFarm bugs compared to those generated by alternative approaches?

RQ4: Performance. How long does it take to generate and validate bugs using BugFarm and other approaches?

A. Experiment Setup and Data Availability

Mutant Generation. We compare BugFarm with two most recent mutant generation techniques, μBERT [25] and LEAM [44]. To inject mutants, μBERT selects AST nodes representative of program behavior—literals, identifiers, expressions, assignments, object fields, method calls, array access, and types. Then, it replaces the tokens in selected AST nodes with the special token `<mask>` and uses CodeBERT to predict the masked token. The intuition is that if CodeBERT predicts a token different from the original one, the transformation introduces a bug. LEAM is a deep learning-based technique that learns to mutate code from large examples of real-world bugs. To that end, they represent code as a sequence of AST nodes and learn to apply eight grammar rules to select and modify the code. Both μBERT and LEAM claim to generate better bugs (mutants confirmed by test execution) compared to classic approaches, namely PIT [12] and Major [23]. Therefore, we did not include classic techniques in our evaluation.

For BugFarm, we used the threshold value $k = 10\%$, mainly because alternative approaches do not change more than a handful of statements in a given code (more details in §V-B). Our experimental results in the rest of this section confirm that even with such a low threshold, we still surpass other techniques in generating hard-to-detect (§V-C) and repair (§V-D) bugs. A higher threshold will make our bugs more complex compared to other approaches, improving margins. We also configured BugFarm to generate at most three mutants per method ($N = 3$ in Algorithm 2) as our preliminary results show that with a higher number, the generated bugs for $k = 10\%$ are almost identical. With a higher threshold value, one can also increase N and generate more diverse bugs. The current implementation of BugFarm's *Bug Generator* component uses GPT-3.5-turbo due to its accessibility and effectiveness at a reasonable cost (\$0.0004 per mutant on average). Using a more advanced LLM such as GPT-4 likely yields better results.

Automated Bug Confirmation. BugFarm and alternative approaches generate mutants, which entails implementing a confirmation process to ensure syntactical correctness and rule out equivalent mutants. Following the related work [44], [25], our confirmation procedure follows three steps: (1) running existing test suites on the original project and selecting the green tests; (2) compiling the generated mutants from the methods covered by the green tests and discarding syntactically incorrect ones; and (3) re-executing previously passed tests on syntactically correct mutants and choosing those killed through test execution. Due to the possibility of generating mutants that are not equivalent but survive test execution, we further performed a manual investigation on a subset of survived mutants. This will ensure the fairness of the experiments and the generalizability of the claims in the paper. All the experiments were conducted with killed mutants, which we refer to as confirmed bugs. The mutant generation techniques generated a total $1,908,566$ mutants, out of which $699,575$ were syntactically correct, and $434,215$ identified as confirmed bugs through testing (details in §V-B).

Manual Bug Confirmation. From the $265,360$ syntactically correct survived mutants, we sampled 300, controlling equal contributions from different sets of generated mutants. We then crowd-sourced the task of *mutant evaluation* to 97 individuals with varying software engineering expertise, from beginner to expert with at least two years of industry experience. We provided each participant with the mutant and original codes and asked them to label mutants with one of *Yes* (the mutant is a bug), *No* (the mutant is equivalent), or *Maybe* (uncertain whether the mutants is equivalent) labels. Individuals monitored their time and reported the amount spent on the task. We also asked them to evaluate how challenging it was to decide on the assigned mutants' labels. Each mutant was labeled by *three* individuals, and we decided on the final label based on the majority vote.

The labeling process took 290 person-hours and participants delivered 900 labels (487 Yes, 384 No, and 29 Maybe). The majority voting provided us with 143 additional confirmed bugs, where 72 (50.35%), 53 (37.06%), and 18 (12.59%) of them belong to BugFarm, μBERT, and LEAM, respectively. During the labeling, we asked participants to identify the difficulty level for labeling each mutant, i.e., whether it was challenging for them to identify the mutant was equivalent or not. Figure 3 shows that BugFarm's mutants were more challenging than other approaches for the participant to label. We believe that this large-scale human study accounts for the threat to the validity of choosing killed mutants as confirmed bugs.

Bug Prediction Models. When selecting bug prediction

[1]The natural language part of our prompts is more complex than the example here. The readers can refer to our artifact to see the exact prompts [27]

[2]The detailed results and additional research questions, e.g., ablation study demonstrating the necessity of prompt crafting, are publicly available [27].

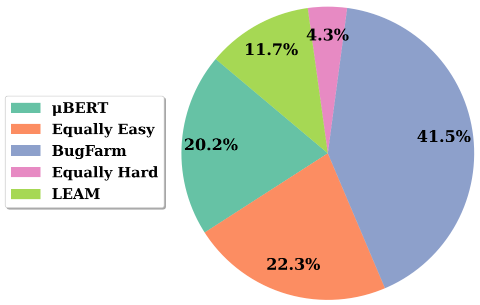


Fig. 3: Evaluating the complexity of generated mutants by human subjects

models, we had to consider the following criteria: the current implementation of BugFarm's *Attention Analyzer* works on transformer-based models and requires the availability of models (weights and internals) to perform the attention analysis (we will consider including closed-source models as future work). We could not find any custom bug prediction model publicly or per request available with such characteristics. Consequently, we chose three pre-trained code-language models that are widely used by the research community for fine-tuning, namely CodeBERT [15], CodeT5 [48], and NatGen [9][3]. We fine-tuned these models using real-world bugs and evaluated the effectiveness of the outcome in predicting bugs generated by different techniques.

CodeBERT is an encoder-only transformer model based on BERT [13] architecture, which is trained on $2.1M$ bi-modal (natural language and code pairs) and $6.4M$ uni-modal (code only) data from CodeSearchNet [19] dataset. The main learning objectives in CodeBERT are Masked Language Modeling (MLM)—the model learns to predict the tokens replaced by a special mask token—and Replaced Token Detection (RTD)—the model learns to detect which token does not belong to the original data. CodeT5 is an encoder-decoder transformer model based on T5 [37] architecture. CodeT5 is trained on $8.35M$ functions from various programming languages provided by CodeSearchNet [19] and BigQuery [1] dataset, and its training objectives include masked span and masked identifier prediction. Such objectives enable the model to understand code semantics better than CodeBERT.

NatGen is also an encoder-decoder transformer model, trained on a generative task of naturalizing source code. Specifically, NatGen starts with CodeT5—based model—parameters and continues the training with a new objective, i.e., re-constructing the original code (natural) given transformed code (de-natural). It uses $8.1M$ pairs of natural and de-natural functions from CodeSearchNet [19] and C/C#. We used the base models of each for our experiments.

Automated Program Repair Model. To evaluate the effectiveness of learning-based techniques in repairing generated bugs, we used FitRepair [49], a state-of-the-art APR technique that outperforms existing approaches. It leverages information retrieval and static analysis to implement domain-specific fine-tuning and prompting strategies. Per the authors' instructions, we used FitRepair with CodeT5-Large in a zero-shot manner to generate the patches for μBERT, LEAM, and BugFarm confirmed bugs and validated patches through test execution.

Subjects. BugFarm is programming-language agnostic; none of its components depend on a specific programming language. However, we chose Java projects in our experiments for the following reasons: (1) LEAM's pre-trained model is on Java, and they have no alternative training dataset for other programming languages; (2) we needed real-world bug datasets for RQ2, and most of the existing real-world bug datasets are in Java. For a fair comparison, we used Defects4J $V2.0$ projects (but used their latest version) as a baseline for bug generation since the alternative approaches are shown to work on them with no issues. The current version of BugFarm supports Maven projects only, so we excluded Mockito and Closure projects from the subjects. The first two columns of Table I show the list of our 15 subjects and the number of methods per subject used for mutant generation.

Evaluation Metrics. To compare the performance of bug prediction models, we use accuracy, precision, recall, F1 score, and False Negative Rate (FNR) as our metrics. To evaluate the APR results, we measure the repair success rate, i.e., the number of bugs the technique successfully patches. *True Positive (TP)* is when the code is buggy, and the model predicts it as buggy. *True Negative (TN)* is when the code is not buggy, and the model predicts it as non-buggy. *False Positive (FP)* is when code is not buggy, but the model predicts it as buggy. *False Negative (FN)* is when the code is buggy, but the model predicts it as non-buggy.

B. RQ1: Characteristics of the Generated Bugs

Table I presents the metrics and their values calculated for the studied approaches (For BugFarm, the reported number is aggregated or averaged across all baseline models). Columns *SCM* and *CB* show the number of *syntactically correct* mutants and *confirmed bugs*, respectively. The mutant generation techniques overall generated a total of $1,908,566$ mutants, out of which $699,575$ were syntactically correct and $434,358$ confirmed as bugs (through test execution and human study). The percentage of syntactically correct to all generated mutants in BugFarm is 85% compared to that of 59.4% and 61.82% for LEAM and μBERT, **corroborating a higher quality of our mutants.** A higher percentage of confirmed bugs (85.55% in BugFarm compared to 59.41% and 61.84% for LEAM and μBERT), along with the human study results presented in §V-A, show the code transformations by BugFarm are more likely to be bugs. The rest of the metrics are computed for confirmed bugs, which we simply refer to as bugs in the rest of the paper:

- *# Statements involved*: This metric indicates the number of statements added, removed, or modified to generate the bugs.

[3]Fine-tuning larger models requires non-trivial computing resources. Our experiments will show that models superior in other code-related tasks (e.g., CodeT5 over CodeBERT) show the significance of BugFarm better. We expect this to hold for larger models as well.

TABLE I: Comparing the characteristics of bugs generated by BUGFARM, μBERT, and LEAM. **M**: # Methods, **SCM**: # Syntactically Correct Mutants, **CB**: # Confirmed Bugs, **SI**: # Statements Involved in bug generation, **LD**: Lines Deleted in bug generation, **ED**: Edit Distance, **OL**: Overlap with LEAM, **OM**: Overlap with μBERT.

Subjects	**M**	**BUGFARM**							**LEAM [44]**					**μBERT [25]**				
		SCM	**CB**	**SI**	**LD**	**ED**	**OL**	**OM**	**SCM**	**CB**	**SI**	**LD**	**ED**	**SCM**	**CB**	**SI**	**LD**	**ED**
cli	276	295	266	2.9	0.0%	33.2	⟨13.9%,0.6⟩	⟨8.6%,0.7⟩	2126	1388	2.6	13.69%	26.1	2053	1532	2.0	0.07%	19.3
codec	901	544	360	3.1	0.0%	31.6	⟨3.1%,0.7⟩	⟨5.5%,0.9⟩	4607	2130	2.8	7.84%	22.4	14738	9551	2.0	0.0%	17.9
collections	4440	3775	3175	2.9	0.0%	29.7	⟨11.1%,0.6⟩	⟨5.1%,0.7⟩	21379	11266	2.8	7.77%	20.9	34038	22196	2.0	0.0%	20.0
compress	4123	573	468	3.0	0.0%	29.4	⟨4.4%,0.7⟩	⟨2.4%,0.8⟩	20401	9373	2.6	12.55%	24.5	57815	23602	2.0	0.0%	24.2
csv	248	282	252	2.7	0.0%	24.9	⟨11.9%,0.7⟩	⟨1.5%,0.7⟩	1803	1189	2.8	14.55%	33.5	131	109	2.0	0.0%	7.0
jxpath	1672	1152	948	3.3	0.0%	28.0	⟨8.0%,0.7⟩	⟨6.1%,0.8⟩	11511	6169	2.9	15.24%	26.2	21335	10907	2.0	0.0%	25.8
lang	3810	4421	3791	3.0	0.0%	30.1	⟨7.5%,0.7⟩	⟨3.1%,0.8⟩	26365	16316	2.7	9.69%	21.4	23312	15732	2.0	0.02%	21.1
math	5796	6466	6207	3.4	0.0%	34.2	⟨4.4%,0.7⟩	⟨2.3%,0.8⟩	43090	42262	2.7	8.43%	17.0	85393	83512	2.0	0.01%	18.3
gson	985	631	545	3.3	0.0%	38.2	⟨7.3%,0.7⟩	⟨1.6%,0.7⟩	4896	2992	2.6	17.95%	27.9	4173	2067	2.0	0.05%	19.8
jackson-core	2626	2281	1732	4.0	0.0%	43.1	⟨2.9%,0.7⟩	⟨2.3%,0.7⟩	21472	9963	2.5	22.58%	25.8	19118	9310	2.1	0.0%	21.6
jackson-db	8076	4828	3965	3.9	0.0%	41.9	⟨7.2%,0.7⟩	⟨6.9%,0.7⟩	36155	18454	2.8	12.17%	28.7	51032	22454	2.0	0.0%	22.3
jackson-xml	586	312	258	3.9	0.0%	44.0	⟨6.3%,0.7⟩	⟨5.9%,0.7⟩	3096	1580	2.7	12.85%	31.1	2576	1226	2.0	0.08%	14.7
jfreechart	8602	4051	3186	3.1	0.0%	27.1	⟨7.4%,0.7⟩	⟨1.1%,0.8⟩	40313	17987	2.5	10.64%	19.8	18924	7045	2.0	0.07%	28.2
joda-time	4279	4188	3734	2.6	0.0%	22.1	⟨8.6%,0.6⟩	⟨3.6%,0.7⟩	26224	15641	2.9	6.29%	24.7	49907	28694	2.0	0.0%	21.1
jsoup	1642	1561	1356	2.8	0.0%	28.5	⟨9.4%,0.7⟩	⟨4.7%,0.7⟩	9534	5456	2.5	14.15%	23.6	6699	4013	2.2	0.05%	21.0
Total	**48062**	**35359**	**30242**	**-**	**-**	**-**	**-**	**-**	**272972**	**162166**	**-**	**-**	**-**	**391244**	**241950**	**-**	**-**	**-**
Average	**3204**	**2357**	**2016**	**3.2**	**0.0%**	**32.4**	**⟨7.6%,0.7⟩**	**⟨4.0%,0.7⟩**	**18198**	**10811**	**2.7**	**12.43%**	**24.9**	**26083**	**16130**	**2.0**	**0.02%**	**20.2**

On average, even with the threshold values of $k = 10\%$, **BUGFARM changes more statements to generate bugs compared to LEAM and μBERT**. Although #SI is higher for BUGFARM bugs, given that these statements are among the least attended statements, we will later see that the models will have a harder time distinguishing them from the correct code (more details in §V-C).

- *# Lines deleted*: Deleting an entire statement is very likely to change attention significantly and result in a runtime exception rather than a semantic bug. Columns *LD* show the percentage of bugs created by only deleting or commenting statements. **BUGFARM does not delete (or comment) any statement through bug-inducing transformation** (0% **on average for all the projects), compared to LEAM** (12.43%) **and μBERT** (0.02**%**).
- *Edit distance*: We also wanted to compare the edit distance between the original and generated bugs. We used Levenshtein [28] edit distance, which measures the minimum number of single-character edits—insertions, deletions, or substitutions—required to change one string into another. The results for edit distance normalized by #CB are available under columns *ED*. **Compared to LEAM and μBERT, BUGFARM's bugs have higher ED values. This indicates that BUGFARM's changes to the code are bigger yet less noticeable, as we will show in RQ2 and RQ3.**
- *Uniqueness*: We were interested to see how much our bugs overlap with those generated by μBERT and LEAM. To that end, we measured the Exact Match (EM) and CodeBLEU [38] values between each BUGFARM bug with all the corresponding bugs generated by μBERT and LEAM. For a given method A, if BUGFARM generates three bugs b_1, b_2, b_3 and LEAM generates four l_1, l_2, l_3, l_4, we construct 12 pairs of $\langle b_i, l_j \rangle$ to compute the EM and CodeBLEU. EM is a strict all-or-nothing metric; being off by a single character results in a score of 0. If the characters of b_i exactly match the characters of b_j, EM = 1 for the pair; otherwise, EM = 0. CodeBLEU is a metric to measure weighted n-gram match between the pairs by considering not just the code tokens but also code syntax via abstract syntax trees (AST) and code semantics via data flow.

  The <EM,CodeBLEU> values are shown under columns *OL* (overlap with LEAM) and *OM* (overlap with μBERT). These numbers show that **only** 7.6% **and** 4% **of the total bugs generated by BUGFARM for all the projects overlap with LEAM and μBERT, respectively**, confirming the uniqueness of BUGFARM bugs. The high number for CodeBLEU (0.7 on average for both) shows that **although BUGFARM bugs are better at challenging learning-based techniques (§V-C-§V-D), they are semantically similar to LEAM and μBERT bugs; potentially as effective as them in their evaluated tasks [44]**.

Summary. Compared to other techniques, BUGFARM's bug-inducing transformations involve more statement modification and change of several code tokens. The overlap between BUGFARM bugs and other approaches is low, demonstrating their uniqueness.

C. RQ2: Effectiveness in Generating Hard-to-Detect Bugs

In this research question, we investigate the effectiveness of bug prediction models on BUGFARM bugs compared to alternative approaches. Given the unavailability of off-the-shelf models as discussed in §V-A, we fine-tuned three pre-trained code-language models (CodeBERT, CodeT5, and NATGEN) using real-world and synthetic bugs. We adapted best practices for fine-tuning transformer models and used the same class distribution in fine-tuning, validating, and testing bug prediction models. All models were fine-tuned for at most 10 epochs with loss-based early-stopping criteria of two consecutive epochs[4]. We selected the model with the least validation error, repeated the evaluation 10 times, and reported the average values.

[4]this is the default setting for fine-tuning CodeT5. To ensure a fair comparison, we checked that all the models converged before 10 epochs.

TABLE II: Effectiveness of studied fine-tuned models in predicting synthetic bugs. Each set of rows shows the same pre-trained model that is fine-tuned on the same dataset but tested on different bug datasets.

	Fine-tune-Model-Test	Acc	Prec	Rec	F1	FNR
1	μBERT-CodeBERT-BugFarm	**71.88 (4.49% ↓)**	**89.94 (1.11% ↓)**	**49.26 (12.18% ↓)**	**63.66 (8.26% ↓)**	**50.74 (15.55% ↑)**
2	μBERT-CodeBERT-LEAM	75.26	90.95	56.09	69.39	43.91
3	LEAM-CodeBERT-BugFarm	**69.42 (10.76% ↓)**	**94.05 (1.48% ↓)**	**41.47 (28.93% ↓)**	**57.56 (20.53% ↓)**	**58.53 (40.53% ↑)**
4	LEAM-CodeBERT-μBERT	77.79	95.46	58.35	72.43	41.65
5	μBERT-CodeT5-BugFarm	**75.35 (0.91% ↓)**	**87.45 (0.35% ↓)**	**59.2 (2.18% ↓)**	**70.6 (1.45% ↓)**	**40.8 (3.34% ↑)**
6	μBERT-CodeT5-LEAM	76.04	87.76	60.52	71.64	39.48
7	LEAM-CodeT5-BugFarm	**71.27 (8.90% ↓)**	**89.29 (1.85% ↓)**	**48.35 (22.86% ↓)**	**62.73 (15.48% ↓)**	**51.65 (38.40% ↑)**
8	LEAM-CodeT5-μBERT	78.23	90.97	62.68	74.22	37.32
9	μBERT-NatGen-BugFarm	**74.42 (1.29% ↓)**	**85.07 (0.29% ↓)**	**59.24 (3.41% ↓)**	**69.84 (2.13% ↓)**	**40.76 (5.40% ↑)**
10	μBERT-NatGen-LEAM	75.39	85.32	61.33	71.36	38.67
11	LEAM-NatGen-BugFarm	**69.46 (9.33% ↓)**	**88.35 (2.14% ↓)**	**44.84 (24.80% ↓)**	**59.49 (17.17% ↓)**	**55.16 (36.64% ↑)**
12	LEAM-NatGen-μBERT	76.61	90.28	59.63	71.82	40.37
13	REAL-CodeBERT-BugFarm	**53.02 (3.84% ↓)**	**55.23 (5.20% ↓)**	**31.92 (11.92% ↓)**	**40.46 (9.44% ↓)**	**68.08 (6.78% ↑)**
14	REAL-CodeBERT-μBERT	55.14	58.26	36.24	44.68	63.76
15	REAL-CodeBERT-LEAM	57.62	61.38	41.12	49.25	58.88
16	REAL-CodeT5-BugFarm	**49.02 (4.72% ↓)**	**48.93 (5.03% ↓)**	**44.64 (9.18% ↓)**	**46.69 (7.20% ↓)**	**55.36 (8.87% ↑)**
17	REAL-CodeT5-μBERT	51.45	51.52	49.15	50.31	50.85
18	REAL-CodeT5-LEAM	53.81	53.79	54.05	53.92	45.95
19	REAL-NatGen-BugFarm	**50.75 (0.12% ↓)**	**50.62 (0.10% ↓)**	**61.07 (0.42% ↓)**	**55.36 (0.23% ↓)**	**38.93 (0.67% ↑)**
20	REAL-NatGen-μBERT	50.81	50.67	61.33	55.49	38.67
21	REAL-NatGen-LEAM	53.81	53.01	67.05	59.21	32.9

Generally, one should run BugFarm on each model for bug generation. However, our investigation showed that fine-tuning does not greatly impact the set of LAS (Algorithm 1 in §IV-B). This is consistent with the findings of prior work that shows fine-tuning only changes the attention of the last few layers, not impacting the overall attention of the model [41]. It also shows that many of the generated BugFarm bugs in this study are reusable by other researchers. Consequently, we only generated bugs for the methods per each model whose LAS set was changed compared to the baseline pre-trained model.

1) Fine-tuning on synthetic bugs: It is possible that a learning-based bug prediction technique uses synthetic bugs for training or fine-tuning. So, we first investigate how challenging BugFarm's bugs are, compared to other synthetic bugs, to challenge such models. To that end, we fine-tuned the three pre-trained models on LEAM and μBERT bugs. This provided us with *six* fine-tuned bug prediction models, namely, μBERT-CodeBERT, μBERT-CodeT5, μBERT-NatGen, LEAM-CodeBERT, LEAM-CodeT5, and LEAM-NatGen. We then evaluated each on BugFarm and the other approach, whose bugs were not used for fine-tuning. For example, we evaluated μBERT-CodeBERT on bugs generated by BugFarm and LEAM. This pairwise comparison will show us which techniques generate bugs that are harder to detect.

The rows 1–12 in Table II show the result of this experiment. For Accuracy, Precision, Recall, and F1 score, the lower metric value indicates the approach's superiority (bugs are harder to distinguish from correct code, hence being detected). For FNR, a higher metric value indicates higher FN, showing the model struggles more to detect them. **From these results, we can clearly see that BugFarm bugs always achieve higher values for FNR (average margin of 23.31% (min=3.34%, max=40.53%)) and lower values for the other metrics (average margin of Accuracy=5.95%, Precision=1.2%, Recall=15.73%, and F1-score=10.84%).**

Furthermore, F1 score values suggest the models fine-tuned on LEAM have a harder time detecting BugFarm bugs than those fine-tuned on μBERT. We believe this is because μBERT bugs are more diverse due to changing many tokens of the code and combining them through beam search, compared to LEAM bugs that try to mimic the bugs scrapped from the GitHub issue trackers.

Note that we have not used BugFarm bugs for fine-tuning models. The reason is BugFarm bugs are in-distribution samples, and alterations are within the decision boundaries for models: we only accept mutants that do not change the model's attention so that the code representation of buggy and correct code is similar (Algorithm 2 Lines 11–14). Fine-tuning on such examples theoretically cannot improve in-distribution performance and may even worsen out-of-distribution performance [26]. As we claimed before, BugFarm bugs are for evaluating and challenging learning-based bug-related tasks and should be used for that purpose only.

2) Fine-tuning on real-world bugs: To avoid any bias in our conclusion based on synthetic bugs, we also fine-tuned baseline models with real-world bugs from three datasets, namely Defects4J [24][5], BugSwarm [45], and RegMiner [42]. The real-world evaluation dataset consists of 723, 3285, and 36412 original and buggy methods from Defects4J, BugSwarm, and RegMiner, respectively. This results in three bug prediction models, i.e., REAL-CodeBERT, REAL-CodeT5, and REAL-NatGen. We evaluated each model on BugFarm, μBERT, and LEAM bugs. The rows 13–21 in Table II show the results of this experiment, with margins indicating the difference with respect to second-best synthetic bug dataset. **These results show similar trends we observed with models fine-tuned on synthetic bugs, i.e., BugFarm bugs result in higher FNR (54.12% on average) and lower Accuracy (50.93% on average), Precision (51.59% on average), Recall (45.88%**

[5]We could use the bugs from Mockito and Closure projects, which were excluded from our subjects since BugFarm only supports Maven at this time

TABLE III: Effectiveness of FitRepair in repairing synthetic bugs. **SI**: Statements involved.

	LEAM [44]	**μBERT [25]**	**BugFarm-CodeBERT**	**BugFarm-CodeT5**	**BugFarm-NatGen**
Total Bugs (SI=1,SI=2,SI>2)	200 (174,25,1)	200 (187,13,0)	200 (125,40,35)	200 (92,43,65)	200 (104,30,66)
Success Rate	36%	49%	29%	29%	28%

on average), and F1 score (47.5% on average) values.

That models fine-tuned on real-world bugs underperform those fine-tuned on synthetic bugs across all metrics. A possible justification for this observation is the distribution shift between real-world bugs (used for fine-tuning) and synthetic bugs (used for testing). The two root causes for these distribution shifts are (1) the real-world bugs belonging to projects *different* than those from which we generated the synthetic bugs; and (2) the nature of real-world bugs is different from synthetic bugs. Especially for BugFarm and μBERT, the bug generation objective does not include any similarity to real-world bugs.

Summary. Bug prediction models, regardless of whether the fine-tuning dataset is from a synthetic or real-world bug dataset, have a harder time detecting BugFarm bugs than other synthetic bugs.

D. RQ3: Effectiveness in Generating Hard-to-Repair Bugs

To further demonstrate the complexity of our bugs, we evaluate the ability of FitRepair in repairing BugFarm, μBERT, and LEAM bugs. We configured FitRepair to generate 100 patches [6] per bug and terminate if this takes more than five hours [7]. Given the time-consuming nature of program repair, we evaluated FitRepair on 1000 generated bugs [8], constitutes from 200 sampled confirmed bugs from each bug dataset (LEAM, μBERT, BugFarm-CodeBERT, BugFarm-CodeT5, and BugFarm-NatGen). When sampling, we controlled the selection of bugs from the same methods for all approaches: we sorted them based on the descending order of method size—measured by the number of characters—and picked the top 200. Our rationale is that the potential locations for bug injection increase when methods are longer. As a result, the likelihood of observing different bugs produced by each technique is higher (similar to the illustrative example in §II). The collected bugs are from 14 out of 15 subject projects.

The first row in Table III shows the distribution of generated bugs based on the number of statements involved in bug generation (#SI from RQ1). Most subject bugs from LEAM and μBERT differ in only one line with the original code, while BugFarm bugs are more diverse concerning this metric. Each bug dataset took FitRepair two to seven hours to generate 35 patches, on average. We validated all the generated patches for each bug, and if one of the patches results in a green test suite, we ignore other patches and consider the bug repaired by FitRepair. The validation process of 1000 bugs took over 100 hours. The second row of Table III shows the percentages of bugs that FitRepair successfully patched.

[6]The tool often generates fewer patches than this max allowance value.
[7]Confirmed by FitRepair authors to ensure proper performance.
[8]This number is 2.5 times than the bugs in FitRepair evaluation.

FitRepair successfully repaired 36% and 49% of the LEAM and μBERT bugs, respectively. In contrast, it can only repair 29%, 29%, and 28% of BugFarm bugs generated for each baseline model. This is not surprising, as APR techniques are known to perform better in repairing bugs with SI=1. In fact, 87% of correct patches for μBERT and LEAM only differ with the corresponding bug in one line. This value is only 62% for BugFarm, which implies a higher complexity of BugFarm's one-line bugs compared to alternative approaches.

Our deeper investigation of the nature of bugs shows that LEAM and μBERT bugs mostly change the conditional/branch statements. Since repair templates of FitRepair are specifically designed to repair bugs in conditional/branch statements (logical expression in if statement), it can repair LEAM and μBERT bugs better. In contrast, **BugFarm is more creative in introducing bug injection due to the power of LLMs in code synthesis, and it considers locations that the model attends less to for injecting the bugs. Consequently, it can challenge learning-based APR techniques crafted to repair known bug patterns.** By looking at the other patches that fixed bugs with SI>=2, we observed the same pattern, i.e., there was at least one statement changing the conditional statements.

Summary. When applied to the same method, BugFarm generates bugs that are harder to repair by learning-based repair techniques compared to alternative approaches. The power of BugFarm will become more evident when the methods are longer, letting it change multiple locations in the method to introduce the bug.

E. RQ4: Performance

We measured the time required to extract attention weights from the models and the time it takes to prompt the LLM (GPT-3.5-turbo). We also compared the total time for bug generation in BugFarm compared to alternative approaches. All the experiments were performed on a workstation with NVIDIA GeForce RTX 3090 GPUs (24GB GDDR6X memory) and 24 3.50GHz Intel 10920X CPUs (128 GB of memory). Figure 4 shows the results, where the red dashed line indicates the mean value. It takes 68, 69, and 86 milliseconds on average for BugFarm to extract and analyze attention weights from CodeBERT, CodeT5, and NatGen models. Prompting the LLM for every method also takes about 9 seconds on average (*Prompting* box chart at the middle). The prompting time can be affected by multiple factors, including the traffic on the model and prompt size, i.e., the number of tokens in the prompt.

We also measured the total time each technique takes to generate mutants (unconfirmed bugs) per method (*End-to-End Mutant Generation* in Figure 4). Compared to attention analysis and prompting, the overhead of other sub-components

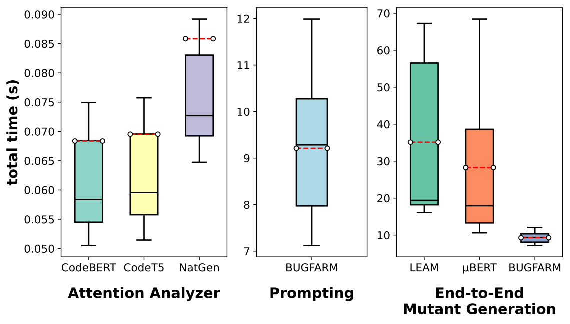


Fig. 4: Performance of BugFarm compared to alternative approaches in mutant generation

in BugFarm, such as parsing and bug selection, is negligible. **Therefore, the average time for generating all the BugFarm mutants per each method is** 9.29 **seconds (mostly dominated by the prompting time). In comparison, it takes LEAM and** μ**BERT** 35 **and** 28 **seconds on average to generate mutants per method**. This is because these approaches generate more mutants, as illustrated in Table I under *#SCM* columns. For LEAM, this time does not include the training time of the model, which is 24 hours [9]. Also, since μBERT takes a long time to mutate big classes, we put a timeout of 15 minutes on it to avoid a long generation time. Automated validation was also very time-consuming; we spent around $60,000$ CPU core-hours to validate over 1.9 million mutants from μBERT, LEAM, and BugFarm.

Summary. BugFarm is an efficient and scalable technique for generating bugs, 74% and 67% faster in end-to-end mutants generation than LEAM and μBERT.

F. Discussion

BugFarm aims to generate hard-to-detect and repair bugs concerning learning-based techniques. Our bugs are not designed to replace real-world bugs but complement them from a new perspective, i.e., having a very close code representation as the original code while different in several places. We have no claim that BugFarm bugs mimic real-world bugs (because they do not need to). As a result, comparing with real-world bugs is out of the scope of this paper. However, BugFarm leverages GPT-3.5-turbo for generating bugs, which theoretically have seen many real-world bugs during training. This can potentially help BugFarm bugs be similar to real-world bugs compared to μBERT, which does not concern such similarity.

VI. Related Work

Real-world bug/vulnerability benchmarks. Several attempts have been made to construct real-world bug datasets manually. Defects4J [24], BugSwarm [45], Bugs.jar [39], and RegMiner [42] are the commonly used Java bug datasets that have mined GitHub to collect regression bugs from bug-fixing commits. BigVul [14] and CVEFixes [4] are real-world examples of vulnerabilities collected from bug-fixing reports in the CVE/NVD database [6]. BugFarm complements the bugs and vulnerabilities in these datasets with hard-to-repair and hard-to-detect bugs. Also, the bugs in these datasets only represent human mistakes, which could be potentially different from the mistakes AI programming tools make.

[9]This number is quoted from their paper.

Learning-based bug/vulnerability generation. Learning-based bug generation was proposed to overcome the limitations of manual defect model construction [7]. Such techniques learn the bug or vulnerability patterns from real-world bug fixes and generate mutants accordingly. DeepMutation [46] is a technique that relies on sequence-to-sequence neural machine translation for learning and generating bugs. SemSeed [36] extracts bug patterns from real-world bug fixes and injects them into other programs so that the bug in the new program is syntactically different but semantically similar. MutationMonkey [3] mines bug patterns from historical changes and transforms them into mutation operators semi-automatically. VulGen [34] combines pattern mining and deep learning to generate realistic bugs. LEAM [44] learns to mutate code from large examples of real-world bug-fixing commits. μBERT [25] produces buggy versions by replacing code tokens with the spacial `<mask>` token, and uses CodeBERT to predict the masked token. Both LEAM and μBERT incorporate beam search [16] to generate bugs that involve more than one statement.

BugFarm is superior to prior learning-based bug-generation techniques in several ways: BugFarm does not involve any training or fine-tuning effort to learn bug patterns and generate bugs. Consequently, it is independent of existing real-world bug datasets or a corpus of bug-fixing commits. Second, while the majority of prior work only generates one-line bugs, BugFarm can be configured to generate bugs that involve multiple statements. Third, BugFarm is the first technique that targets the generation of bugs that can challenge learning-based bug detectors and repair tools, or bugs that represent the AI programming tools' mistakes, rather than human mistakes. Our empirical evaluation confirmed that these properties result in the generation of bugs that are hard-to-detect and hard-to-repair.

VII. Concluding Remarks

Bug benchmarks are essential in software engineering to evaluate automated techniques concerning bugs. The advent of learning-based bug-related techniques demands automated bug-generation techniques for proper evaluation. In this paper, we presented BugFarm, a model-in-the-loop technique for the automated generation of hard-to-detect and repair bugs. Our empirical evaluation shows the superiority of BugFarm to alternative mutant generation approaches in generating unique and high-quality hard-to-detect and repair bugs. BugFarm does not rely on existing bug datasets and is model-agnostic.

Acknowledgments

The NSF CCF-2238045 grant supports this work. We thank Prof. Darko Marinov for sharing their wisdom to improve this research, the UIUC+'24 program students who helped with the human study and manual verification of bugs, and the Illinois National Center for Supercomputing Applications (NCSA) for providing computing resources.

References

[1] Bigquery dataset. https://cloud.google.com/bigquery/public-data, 2025.

[2] O. AI. Open ai chatgpt. https://openai.com/blog/chatgpt, 2025.

[3] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer. What it would take to use mutation testing in industry—a study at facebook. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 268–277, 2021.

[4] G. Bhandari, A. Naseer, and L. Moonen. Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2021, page 30–39, New York, NY, USA, 2021. Association for Computing Machinery.

[5] M. Böhme and A. Roychoudhury. Corebench: studying complexity of regression errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 105–115, New York, NY, USA, 2014. Association for Computing Machinery.

[6] H. Booth, D. Rike, and G. A. Witte. The national vulnerability database (nvd): Overview. 2013.

[7] D. B. Brown, M. Vaughn, B. Liblit, and T. Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 511–522, New York, NY, USA, 2017. Association for Computing Machinery.

[8] BugsInPy. Bugsinpy: Dataset of real-world python bugs. https://github.com/soarsmu/BugsInPy, 2025.

[9] S. Chakraborty, T. Ahmed, Y. Ding, P. T. Devanbu, and B. Ray. Natgen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 18–30, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Y. Chen and R. Jabbarvand. Neurosymbolic repair of test flakiness. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1402–1414, 2024.

[11] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[12] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. Pit: a practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 449–452, New York, NY, USA, 2016. Association for Computing Machinery.

[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In J. Burstein, C. Doran, and T. Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[14] J. Fan, Y. Li, S. Wang, and T. N. Nguyen. A c/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 508–512, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. CodeBERT: A pre-trained model for programming and natural languages. In T. Cohn, Y. He, and Y. Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, Nov. 2020. Association for Computational Linguistics.

[16] M. Freitag and Y. Al-Onaizan. Beam search strategies for neural machine translation. In T. Luong, A. Birch, G. Neubig, and A. Finch, editors, *Proceedings of the First Workshop on Neural Machine Translation*, pages 56–60, Vancouver, Aug. 2017. Association for Computational Linguistics.

[17] GitHub. Github copilot. https://github.com/features/copilot, 2025.

[18] M. Group. Manysstubs4j dataset. https://github.com/mast-group/mineSStuBs, 2025.

[19] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

[20] A. R. Ibrahimzada, K. Ke, M. Pawagi, M. S. Abid, R. Pan, S. Sinha, and R. Jabbarvand. Alphatrans: A neuro-symbolic compositional approach for repository-level code translation and validation. *Proceedings of the ACM on Software Engineering*, 2(FSE):2454–2476, 2025.

[21] A. R. Ibrahimzada, Y. Varli, D. Tekinoglu, and R. Jabbarvand. Perfect is the enemy of test oracle. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 70–81, New York, NY, USA, 2022. Association for Computing Machinery.

[22] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379–1393, 2009. Source Code Analysis and Manipulation, SCAM 2008.

[23] R. Just. The major mutation framework: efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 433–436, New York, NY, USA, 2014. Association for Computing Machinery.

[24] 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*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery.

[25] A. Khanfir, R. Degiovanni, M. Papadakis, and Y. L. Traon. Efficient mutation testing via pre-trained language models. *arXiv preprint arXiv:2301.03543*, 2023.

[26] A. Kumar, A. Raghunathan, R. M. Jones, T. Ma, and P. Liang. Fine-tuning can distort pretrained features and underperform out-of-distribution. In *International Conference on Learning Representations*, 2022.

[27] I. C. Lab. Bugfarm artifact website. https://github.com/Intelligent-CAT-Lab/BugFarm, 2025.

[28] V. I. Levenshtein et al. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710. Soviet Union, 1966.

[29] X. Li, S. Zhu, M. d'Amorim, and A. Orso. Enlightened debugging. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 82–92, 2018.

[30] C. Liu, Y. Chen, and R. Jabbarvand. Codemind: Evaluating large language models for code reasoning. *arXiv preprint arXiv:2402.09664*, 2024.

[31] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, volume 5. Chicago, Illinois, 2005.

[32] Meta. Meta llama. https://ai.facebook.com/blog/large-language-model-llama-meta-ai/, 2025.

[33] T. Nguyen and E. Wong. In-context example selection with influences. *arXiv preprint arXiv:2302.11042*, 2023.

[34] Y. Nong, Y. Ou, M. Pradel, F. Chen, and H. Cai. Vulgen: Realistic vulnerability generation via pattern mining and deep learning. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2527–2539, 2023.

[35] R. Pan, A. R. Ibrahimzada, R. Krishna, D. Sankar, L. P. Wassi, M. Merler, B. Sobolev, R. Pavuluri, S. Sinha, and R. Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[36] J. Patra and M. Pradel. Semantic bug seeding: a learning-based approach for creating realistic bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 906–918, New York, NY, USA, 2021. Association for Computing Machinery.

[37] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21(1), Jan. 2020.

[38] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

[39] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 10–13, New York, NY, USA, 2018. Association for Computing Machinery.

[40] R. K. Samala, H.-P. Chan, L. Hadjiiski, and S. Koneru. Hazards of data leakage in machine learning: a study on classification of breast cancer using deep neural networks. In *Medical Imaging 2020: Computer-Aided Diagnosis*, volume 11314, pages 279–284. SPIE, 2020.

[41] E. Shi, Y. Wang, H. Zhang, L. Du, S. Han, D. Zhang, and H. Sun. Towards efficient fine-tuning of pre-trained code models: An experimental study and beyond. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 39–51, New York, NY, USA, 2023. Association for Computing Machinery.

[42] X. Song, Y. Lin, S. H. Ng, Y. Wu, X. Peng, J. S. Dong, and H. Mei. Regminer: towards constructing a large regression dataset from code evolution history. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 314–326, New York, NY, USA, 2022. Association for Computing Machinery.

[43] R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[44] Z. Tian, J. Chen, Q. Zhu, J. Yang, and L. Zhang. Learning to construct better mutation faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ASE '22, New York, NY, USA, 2023. Association for Computing Machinery.

[45] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 339–349, 2019.

[46] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 301–312, 2019.

[47] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin. Attention is all you need. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[48] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708, Online and Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics.

[49] C. S. Xia, Y. Ding, and L. Zhang. The plastic surgery hypothesis in the era of large language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 522–534, 2023.

[50] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 1–10, New York, NY, USA, 2022. Association for Computing Machinery.

[51] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 17–26, 2015.