

Automated Bug Generation in the era of Large Language Models

ALI REZA IBRAHIMZADA, University of Illinois Urbana-Champaign, USA

YANG CHEN, University of Illinois Urbana-Champaign, USA

RYAN RONG*, The Peddie School, USA

REYHANEH JABBARVAND, University of Illinois Urbana-Champaign, USA

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 are conflicting for 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 be detected.

We propose BUGFARM, to transform arbitrary code into multiple complex bugs. BUGFARM leverages Large Language Models (LLMs) to mutate code in multiple locations (hard-to-repair). To ensure that multiple modifications do not notably change the code representation, BUGFARM analyzes the attention of the underlying model and instructs LLMs to only change the least attended locations (hard-to-detect). Our comprehensive evaluation of 320k+ bugs from over 2.5M 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 41% higher False Negative Rate and 11%, 6%, 29%, and 21% lower Accuracy, Precision, Recall, and F1 score) and hard to repair by state-of-the-art learning-based program repair technique (22% repair success rate compared to 34% and 49% of LEAM and μ BERT bugs). BUGFARM is efficient, i.e., it takes nine seconds for it to mutate a code without any prior training time.

1 INTRODUCTION

Problem Statement and Significance. Machine learning is becoming the new automation technology, and software engineering automation is no exception. AI pair programming techniques such as Copilot [24] are integrated into IDEs, and LLMs such as GPT-4 [48] and Bard [25] are now accessible through API. Following this trend, state-of-the-art software analysis techniques either fine-tune pre-trained code-language models or prompt LLMs to automate code-related tasks¹. This demands automated techniques for generating high-quality datasets to assess their true effectiveness. Concerning bug-related tasks such as bug prediction, localization, and repair, 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 [9, 40]. A similar definition

*Work done when Ryan Rong was an intern at the University of Illinois Urbana-Champaign.

¹We differentiate between LLMs and code-language models. The former models are large (> 10B parameters) and are used through zero- or few-shot prompting. The latter models are smaller and are used after fine-tuning.

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 [68] or to check if the code representation is closer to buggy or correct examples they have seen [30]. As a result, 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, making debugging techniques struggle to localize and understand their interactions [38] to fix them automatically. A similar definition holds from the learning-based software engineering, but with a slight consideration: The power of machine learning is in learning from patterns that persist in their dataset. Consequently, modifying a code in multiple locations, where the learning-based debugging techniques specifically look to localize or repair the bug, would be counter-intuitive. Can existing techniques provide complex bugs, as defined, to evaluate learning-based techniques?

Research Gap. Several attempts have been made to construct real-world bug datasets, e.g., Defects4J [34], BUGSWARM [60], BugsInPy [13], RegMiner [57], and ManySStuBs4J [2]. While representing real-world bugs, such datasets are only limited to the projects and bug patterns included in their mining efforts. More importantly, they may be included in the training/fine-tuning data of learning-based techniques, causing data leakage [54]. To overcome these limitations, researchers have proposed automated bug generation techniques [32, 36, 59]. Such techniques, by default, change a few locations in the code (easy to repair and detect) but can be configured to modify multiple lines. That said, changing multiple lines randomly [36] or following a set of heuristics [59] may result in mutants with a different code representation than the correct code, making them easy to detect. Furthermore, changing lines that learning-based repair techniques have been trained to look into and repair [66], such as *conditional statements*, results in bugs that can be easily repaired, even if they involve multiple lines.

Proposed Solution. To advance automated bug generation concerning the evaluation of learning-based bug-related tasks, we propose BUGFARM. For a given arbitrary code, BUGFARM prompts a Large Language Model (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 too many locations does not drastically change the code representation of the mutant, BUGFARM analyzes the attention of the underlying model and instructs LLMs to only change those the model attends least to. As a result, the generated code has a similar representation to the original one but differs within multiple locations. This makes them hard to detect by bug prediction approaches and hard to repair by debugging techniques. BUGFARM is programming language-agnostic, i.e., making it a pragmatic approach to generate bugs for any programming language. Like related mutant generation techniques, BUGFARM creates unconfirmed bugs, which should be validated through test execution. Our notable contributions are:

- **A new perspective into bug generation:** We propose a novel technique for bug generation concerning the evaluation of the learning-based bug-related tasks. Our proposed technique, BUGFARM, relies on the model’s attention to different parts of code to identify where to inject bugs so that the impact of changes on code representation is minimal. It leverages such information to craft contextual prompts for LLMs to mutate any arbitrary code. The implementation and artifact of BUGFARM, including all the generated bugs, are publicly available [5].
- **Empirical evaluation:** We extensively evaluated BUGFARM and two most recent learning-based bug generation approaches, LEAM [59] and μ BERT [36]. Our evaluation of 320k+ bugs (from over 2.5M mutants) generated for 15 Java projects answers research questions related to (1) the properties of generated bugs, (2) the ability of learning-based bug prediction models to detect

```

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

```

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 generated by LEAM [59]

```

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 μ BERT [36]

```

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 null;
}

```

(d) Bug generated by BUGFARM

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

them, (3) the effectiveness of a learning-based program repair technique to repair them, (4) the impact of prompt crafting, and (5) the performance characteristics of techniques. The results confirm that BUGFARM bugs are hard-to-detect (up to 41% higher False Negative Rate and 11%, 6%, 29%, and 21% lower Accuracy, Precision, Recall, and F1 score) and hard-to-repair (22% repair success rate compared to 34% and 49% of LEAM and μ BERT bugs). Also, generating effective bugs is more efficient using BUGFARM.

2 ILLUSTRATIVE EXAMPLE

To illustrate the limitations of prior work and present the key idea behind BUGFARM, we will use the code snippets in Figure 1. The original code in Figure 1a takes `bag` and `transformer` objects as inputs. If `bag` is not empty, it moves all the objects inside it to `transformer`. There are several ways to inject bugs into this code snippet by adding, deleting, or modifying the 13 statements in its body (some statements are split into two lines for better presentation). These bugs are intended to challenge bug prediction and program repair models we used in our evaluation (§5), which are fine-tuned on top of CodeT5 [64]. Concerning the attention of the underlying model to the original code, the two least attended statements (details in algorithm 1) are highlighted in green.

We leverage three bug generation tools, namely, BUGFARM, LEAM [59], and μ BERT [36] to transform the original code snippets into bugs (bugs were confirmed through test execution). LEAM is a technique that represents code as a sequence of AST nodes and learns to apply grammar rules to select and modify the code for bug generation. μ BERT 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. Figures 1b and 1c show one of the

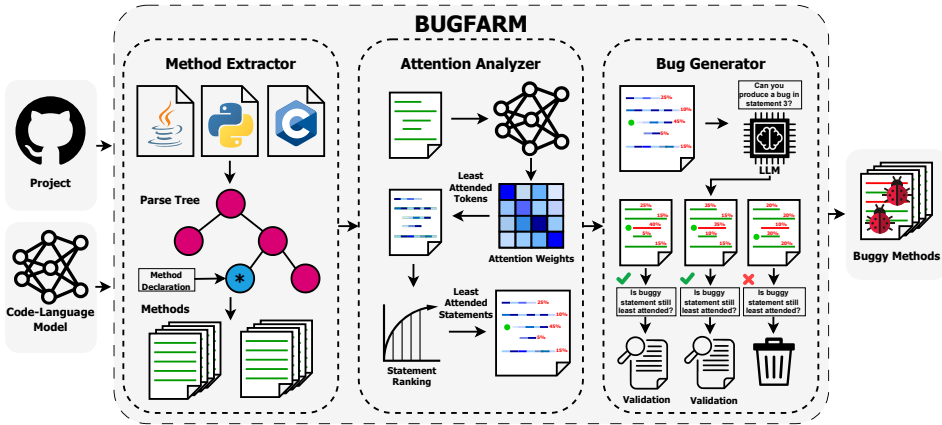


Fig. 2. Overview of BUGFARM

generated bugs by LEAM and μ BERT (LEAM and μ BERT generate 19 and 68 mutants in total, out of which only 6 and 1 are confirmed bugs). BUGFARM generates a bug shown 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 μ BERT bugs modify only one line that is not among the least attended statements. As a result, when running against our studied bug prediction model (§5.3) and program repair model (§5.4), they were easily detected and repaired. In contrast, the same models failed to detect and repair BUGFARM bug.

3 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 then identifies the bug location candidates by analyzing the model’s attention to individual code tokens. To generate bugs for each method, BUGFARM crafts a prompt to an LLM, including the code to be transformed into bugs and additional contexts reflecting candidate locations for bug injection. From the generated bugs, BUGFARM selects those that are syntactically correct, where the bug-inducing changes do not significantly change the attention of the model.

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 bug injection (more details in §4.1).

Recent state-of-the-art learning-based software engineering models are all transformer-based due to the naturalness of code [28] and the promising results of such models in natural language processing (NLP) [62]. Transformer-based models rely on attention for neural code representation. So, to identify the candidate locations to inject unnoticeable bugs into a given method, *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 (more details in §4.2).

Finally, *Bug Generator* component takes the list of bug location candidates as input and creates a prompt consisting of natural language instruction, bug location candidates, and the method. It then sends the prompt to an LLM and collects the generated bugs in the response. For the generated bugs by LLM, BUGFARM computes the extent to which the bug-inducing changes impact the model’s attention to the buggy method compared to the input method, and selects those with negligible impact on the attention. It also discards the bugs that are syntactically incorrect for the sake of quality (more details in §4.3).

4 BUGFARM

In this section, we will explain the details of three main components of BUGFARM. Specifically, we will first discuss how BUGFARM parses the subjects and extract 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?

4.1 Method Extractor

Our approach sets itself apart from most prior work, which relies on defect datasets for automated bug generation [46, 50, 59]. Instead, it focuses on injecting diverse and complex bugs into *any* well-maintained project using LLMs, regardless of the programming language. As a result, we assume that the input to the BUGFARM is a project rather than a single method. As a first step, BUGFARM first requires extracting the implemented methods in each input project. To that end, *Method Extractor* component leverages a language-specific 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 both signatures and body (excluding docstrings) using *MethodDeclaration* and *ConstructorDeclaration* blocks in the parse tree. Each method will be passed to the *Attention Analyzer* component, which we will describe next.

4.2 Attention Analyzer

State-of-the-art code-language models are based on the Transformer architecture [62]—an encoder to capture input’s contextual representations and a decoder to generate output tokens. To produce contextualized vector representation of a sequence of tokens, Transformers rely on *Multi-Head Self-Attention*. More specifically, for a method that consists of n tokens $\overrightarrow{Tkn} = \{m_0, \dots, m_{n-1}\}$, a Transformer model with \mathcal{L} layers takes \overrightarrow{Tkn} as input and produces $\mathbf{H}^\ell = [\mathbf{h}_0^\ell, \dots, \mathbf{h}_{n-1}^\ell]$. Here, \mathbf{H}^ℓ corresponds to the hidden vector representations in layer $\ell \in \{1, 2, \dots, \mathcal{L}\}$. In each Transformer layer ℓ , multiple self-attention heads are used to aggregate the outputs of the previous layer. Consequently, for each token m_i in the method, the self-attention assigns a set of attention weights with respect to all other tokens in the input sequence, i.e., $\text{Attention}(m_i) = \{\alpha_{i0}, \dots, \alpha_{in-1}\}$, where α_{ij} indicates the relative attention of m_i to m_j .

The hidden representation weights of Transformers will be computed based on the attention weights, making the attention weights crucial to identify the importance of each token in the final representation of the method. Recent research has shown that attention analysis can be utilized to interpret the knowledge learned during training [27, 30, 63]. Following their insights, BUGFARM leverages self-attention weight analysis to scrutinize the input method and 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 method

to be transformed into a bug, 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).

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 amount of overlap between least attended tokens should achieve a lower score, thus, be considered the least attended statement. Without normalizing the statement length, longer statements will be penalized, and BUGFARM never selects them as a bug injection location. 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).

Algorithm 1: Attention Analyzer

Inputs: Method *method*, Threshold k , Transformer-based model M

Output: Least attended statements LAS

```

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

```

4.3 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, very likely, will 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 easier. 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.

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 [43], PaLM [26], Copilot [24], Alpaca [58], and ChatGPT [3]. The only consideration is to check if the size of the prompt matches the context window of the LLM.

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      $\lfloor$  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       $\lfloor$  check  $\leftarrow$  false;
14       $\lfloor$  break;
15  if check then
16     $\lfloor$  Bugs  $\leftarrow$  Bugs  $\cup$  Response;

```

After the LLM’s response to the prompts, *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. Ideally, we want the produced bugs to be syntactically correct. To that end, our prompts also include instructions to produce syntactically correct code. However, our experiments showed that despite asking an LLM to produce syntactically correct code, specifically for longer methods, it generates responses that are not parsable. Furthermore, our goal is to generate bugs that have a very close neural representation with the

original code. So, we assess the model’s attention to the generated bugs and only accept them if the changes are still among the least attended statements.

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 [45, 67]. 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)². 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).

5 EVALUATION

To evaluate the effectiveness of BUGFARM, we compare it with two state-of-the-art automated bug generation approaches, namely LEAM and μ BERT, investigating the following research questions:

- RQ1: Characteristics of the Generated Bugs.** Can BUGFARM successfully inject bugs into any arbitrary code? What are the characteristics of the generated bugs by each approach? To what extent do our bugs overlap with those generated by other approaches?
- 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 can a learning-based Automated Program Repair (APR) technique repairs BUGFARM bugs compared to that generated by alternative approaches?
- RQ4: Necessity of Prompt Engineering.** What are the benefits of prompt engineering in BUGFARM?
- RQ5: Performance.** How long does it take to generate and validate bugs using BUGFARM and other approaches?

5.1 Experimental Setup and Data Availability

Alternative Approaches. We compare BUGFARM with two most recent mutant generation techniques, μ BERT [36] and LEAM [59]. To inject mutants, μ BERT 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 spacial 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 μ BERT and LEAM claim to generate better bugs (mutants confirmed by test execution) compared to classic approaches, namely PIT [16] and Major [33]. 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 §5.2). Our experimental results in the rest of this section confirm that even with such a low threshold, we still surpass other techniques. We expect a higher threshold will make our bugs more complex compared to

²The 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 [5]

other approaches, improving margins. We also configured BUGFARM to generate at most three mutants per method ($N = 3$ in Algorithm 2) due to budget allowance. The current implementation of BUGFARM’s *Bug Generator* component uses GPT-3.5-turbo as an LLM due to its superiority over a large number of models [12, 48, 49, 69] and cost-effectiveness. Using a more advanced LLM such as GPT-4 likely yields better results, not worse.

Bug Validation. Alternative approaches we compare against generate mutant, which may not necessarily change the outcome of test execution. Worse, the mutants may not even be compilable. For BUGFARM, although we ask the LLM to generate syntactically correct bugs, it is still not guaranteed that the generated code executes and changes the outcome of test execution. This entails implementing a validation process to confirm if the generated mutants/bugs are compilable and can change the outcome of test execution to consider them bugs. Our validation procedure follows these steps: We first run existing test suites on the original code and select the green tests. Next, we take all the generated mutants/bugs by different techniques, compile them, and run the previously green tests against them (each mutant will be tested in isolation). We discard those that do not compile or result in no failure after test execution. All the experiments answering RQ1-RQ5 are done with confirmed bugs.

Bug Prediction Models. When selecting bug prediction models, we had to consider the following criteria: (1) the current implementation of BUGFARM’s *Attention Analyzer* component works on transformer-based models, including customized trained models and those fine-tuned on pre-trained code-language models; (2) we require the availability of models to perform the attention analysis, so closed-source models such as Codex [15] are off the table; (3) to compare against other bug prediction approaches, we require models with either pre-trained models or required artifacts for training available. 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, namely CodeBERT [22], CodeT5 [64], and NATGEN [14]³. We fine-tuned these models using the real world and synthetic bugs, i.e., those generated by LEAM and μ BERT, and compared their effectiveness in predicting our bugs.

CodeBERT is an encoder-only transformer model based on BERT [20] architecture, which is trained on 2.1M bi-modal (natural language and code pairs) and 6.4M uni-modal (code only) data from CodeSearchNet [29] 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 [51] architecture. CodeT5 is trained on 8.35M functions from various programming languages provided by CodeSearchNet [29] 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 [29] and C/C#. We used the base model of CodeBERT, CodeT5, and NATGEN for our experiments.

Automated Program Repair Model. To evaluate the effectiveness of learning-based techniques in repairing generated bugs, we used FitRepair [66]. FitRepair is a state-of-the-art APR technique that

³Fine-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 1. Comparing the characteristics of bugs generated by BUGFARM, μ BERT, and LEAM. **M**: # Methods, **UB**: # Unconfirmed Bugs, **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 μ BERT.

Subjects	M	BUGFARM						LEAM [59]					μ BERT [36]					
		UB	CB	SI	LD	ED	OL	OM	UB	CB	SI	LD	ED	UB	CB	SI	LD	ED
cli	276	438	263 (60%)	2.86	0%	32.14	<14%,0.64>	<8%,0.67>	3959	1375 (34.73%)	2.56	13.82%	26.2	12641	1528 (12%)	2.00	0.07%	19.38
codec	901	1586	357 (23%)	3.12	0%	32.39	<3%,0.67>	<4%,0.85>	13603	2114 (16%)	2.59	7.95%	20.33	42529	9541 (22%)	2.01	0%	17.89
collections	4440	7373	2958 (40%)	2.55	0%	27.01	<7%,0.67>	<5%,0.73>	56196	9356 (17%)	2.71	8.87%	21.33	132009	21367 (16%)	2.01	0%	20.02
compress	4123	6544	464 (7%)	3.04	0%	30.29	<5%,0.69>	<2%,0.80>	61488	9085 (15%)	2.57	12.87%	24.56	167259	22657 (14%)	1.97	0%	23.83
csv	248	379	250 (66%)	2.66	0%	24.30	<12%,0.66>	<1%,0.75>	3560	1181 (33%)	2.83	14.65%	33.60	818	106 (13%)	1.98	0%	6.81
jxpath	1672	2601	912 (35%)	3.11	0%	26.09	<6%,0.69>	<5%,0.78>	26615	5645 (21%)	2.86	16.44%	26.81	90153	10678 (12%)	2.00	0%	25.88
lang	3810	6505	3734 (57%)	2.95	0%	29.76	<7%,0.68>	<3%,0.76>	54101	15861 (29%)	2.66	10.01%	21.60	80200	15618 (19%)	2.00	0.02%	21.13
math	5796	8935	6104 (68%)	3.27	0%	33.12	<4%,0.67>	<2%,0.77>	92243	40299 (44%)	2.70	8.56%	17.15	359023	83206 (23%)	2.01	0.01%	18.30
gson	985	1497	537 (36%)	3.19	0%	36.43	<8%,0.7>	<2%,0.68>	14496	2957 (20%)	2.60	18.16%	27.89	25359	2067 (8%)	2.00	0.05%	19.84
jackson-core	2626	4494	1707 (38%)	3.85	0%	41.85	<3%,0.68>	<2%,0.72>	44364	9697 (22%)	2.43	22.74%	25.74	62963	9296 (15%)	2.09	0%	21.58
jackson-db	8076	12812	3855 (30%)	3.76	0%	41.70	<6%,0.66>	<5%,0.74>	115859	17789 (15%)	2.78	12.61%	28.96	254947	22300 (9%)	2.01	0%	22.38
jackson-xml	586	877	249 (28%)	3.80	0%	44.24	<6%,0.68>	<5%,0.71>	9935	1541 (16%)	2.67	13.17%	31.46	16420	1225 (7%)	1.98	0.08%	14.64
jfreechart	8602	13679	3035 (22%)	2.88	0%	25.26	<5%,0.71>	<0.6%,0.81>	164460	16716 (10%)	2.44	11.23%	19.64	111341	7026 (6%)	2.01	0.07%	28.20
joda-time	4279	7833	3574 (46%)	2.38	0%	20.61	<6%,0.66>	<2%,0.74>	69548	13739 (20%)	2.87	7.01%	24.73	141682	28010 (20%)	2.00	0%	21.21
jsoup	1642	2694	1335 (50%)	2.79	0%	28.47	<9%,0.66>	<4.5%,0.7>	22632	5311 (23%)	2.52	14.54%	23.54	30420	4002 (13%)	2.04	0.05%	21.07
Total	48062	78247	29334 (37%)	-	-	-	-	-	753059	152666 (20%)	-	-	-	1527764	238627 (16%)	-	-	-
Average	3204	5216	1956 (-)	3.08	0%	31.58	<7%,0.67>	<3.49%,0.75>	50204	10178 (-)	2.65	13%	24.90	101851	15908 (-)	2.00	0.02%	20.14

outperforms existing approaches. It leverages information retrieval and static analysis to implement domain-specific fine-tuning and prompting strategies. We used FitRepair with CodeT5-Large in a zero-shot manner to generate patches for μ BERT, LEAM, and BUGFARM bugs and validated if the patches thoroughly fix the bugs 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 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 1 show the list of our 15 subjects and the number of methods per subject used for bug generation.

Evaluation Metrics. To compare the performance of bug prediction models, we use accuracy, precision, recall, F1 score, False Positive Rate (FPR), 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. We define TP, TN, FP, and FN as below:

- *True Positive (TP)*. The code is buggy; the model predicts it as buggy.
- *True Negative (TN)*. The code is not buggy; the model predicts it as non-buggy.
- *False Positive (FP)*. The code is not buggy; the model predicts it as buggy.
- *False Negative (FN)*. The code is buggy; the model predicts it as non-buggy.

Data Availability. The implementation of BUGFARM and all the artifacts required for reproducing the results presented in this paper are publicly available [5].

5.2 RQ1: Quality of the Generated Bugs

To answer this research question, we compared the characteristics of the bugs generated by BUGFARM with alternative approaches using the metrics listed below. The metrics are computed for all the methods and averaged for each project. For BUGFARM, we have also averaged the numbers across all the models (CodeBERT, CodeT5, and NATGEN)⁴.

⁴BugFarm data per base model is publicly available in the artifact website.

- *# Unconfirmed bugs*: This metric, computed under columns *UB* in Table 1, shows the total number of unconfirmed bugs/mutants generated for each project. **All the approaches successfully generate mutants/bugs for all the methods in the subject projects. μ BERT generates 1.5 and 8 times more bugs compared to LEAM and BUGFARM, respectively.**
- *# Confirmed bugs*: This metric, shown under columns *CB* in Table 1, measures the number of confirmed bugs per each project generated by different approaches. The numbers inside the parenthesis indicate the percentages of confirmed bugs compared to unconfirmed bugs. A higher percentage implicitly implies the superiority of the technique in bug generation, i.e., the code transformations are more likely to be bugs. **From these results, we see that 37% of the generated bugs by BUGFARM are real bugs, compared to 20% and 16% for LEAM and μ BERT, respectively.** Except for one project (Commons Compress), the #CB/#UB ratio for BUGFARM is higher, with a significant margin. The rest of the metrics are computed for confirmed bugs, which we refer to as bugs for simplicity.
- *# Statements involved in bug generation*: This metric indicates the number of statements added, removed, or modified to generate the bugs. On average, even with the threshold values of $k = 10\%$, **BUGFARM changes more statements to generate bugs compared to LEAM and μ BERT.** 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 §5.3).
- *# Lines deleted*: Deleting an entire statement, while likely to create a bug, may not be the best bug-inducing transformation, as it will change attention significantly and result in a runtime exception rather than a semantic bug. So, we also wanted to compare the generated bugs from this aspect. Columns *LD* in Table 1 show the percentage of bugs created with only lines deleted (or commented) during bug generation. **BUGFARM does not delete (or comment) any statement through bug-inducing transformation (0% on average for all the projects), compared to LEAM (13%) and μ BERT (0.02%).**
- *Edit distance*: We also wanted to compare the edit distance between the original and buggy versions generated by each technique. We used Levenshtein [37] edit distance, which measures the minimum number of single-character edits—insertions, deletions, or substitutions—required to change one string into another. Each code is represented as a string of characters to compute the Levenshtein edit distance. The results for edit distance normalized by #CB are available under columns *ED* in Table 1. **Compared to LEAM and μ BERT, BUGFARM’s bugs have higher ED values. This again indicates that BUGFARM’s changes to the code are bigger yet less noticeable, as we will show in RQ2 and RQ3.** Between μ BERT and LEAM, the latter generates bugs different from the original code in more places.
- *Uniqueness*: Finally, we were interested to see how much our bugs overlap with those generated by μ BERT and LEAM. To that end, we measured the Exact Match (EM) and CodeBLEU [52] values between each BUGFARM bug with all the corresponding bugs generated by μ BERT and LEAM. For example, 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 and CodeBLUE values are shown under columns *OL* (overlap with LEAM) and *OM* (overlap with μ BERT) in Table 1. The first number indicates EM, and the second one is CodeBLEU. These numbers show that **only 7% and 3.49% of the total bugs generated by**

BUGFARM for all the projects overlap with LEAM and μ BERT, respectively, confirming the uniqueness of BUGFARM bugs. The average CodeBLEU values are also 0.67 and 0.75. The high number for CodeBLEU is not a threat here, considering bug transformations do not greatly change the code compared to the original one. In fact, it shows that **although BUGFARM bugs are better at challenging learning-based techniques, they are semantically similar to LEAM and μ BERT bugs; potentially as effective as them in their evaluated tasks [59].**

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.

5.3 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 §5.1, 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 epochs⁵ and selected the model with the least validation error in our experiments. For more reliable results, we repeated the evaluation 10 times and reported the average values.

Generally, one should run BUGFARM on each fine-tuned model to generate bugs accordingly. However, our investigation showed that fine-tuning does not greatly impact the set of LAS (Algorithm 1 in §4.2). 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 [56]. 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.

5.3.1 Fine-tuning on synthetic bugs. We fine-tuned the three baseline pre-trained models on LEAM and μ BERT bugs. This provided us with *six* fine-tuned bug prediction models, namely, μ BERT-CodeBERT, μ BERT-CodeT5, μ BERT-NATGEN, LEAM-CodeBERT, LEAM-CodeT5, and LEAM-NATGEN. Our goal is not only to evaluate these models in predicting BUGFARM bugs, but also to see the extent to which BUGFARM bugs are harder to be detected than other techniques. To that end, we evaluated the fine-tuned bug prediction models on BUGFARM and the other approach, whose bugs were not used for fine-tuning. For example, we evaluated μ BERT-CodeBERT on bugs generated by BUGFARM and LEAM. To ensure a fair comparison, we considered methods per each project that both comparing techniques had generated confirmed bugs for them. This will result in the same FPR values for both techniques, but allows us to focus on the effectiveness of bugs for evaluating the model better. This is important from the security perspective since missing a bug/vulnerability is more severe than marking a correct code as buggy [4].

The rows 1–12 in Table 2 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 the bugs are harder to detect. **From these results, we can clearly see that BUGFARM bugs always achieve higher values for FNR (average margin of 23% (min=2.76%,max=40.89%)) and lower values**

⁵this is the default setting for fine-tuning CodeT5. To ensure a fair comparison, we checked that all the models converged before 10 epochs.

Table 2. Effectiveness of studied fine-tuned models in predicting synthetic bugs. Each distinct set of rows show 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	FPR	FNR
1	μ BERT-CodeBERT-BUGFARM	71.95 (4.40% ↓)	90.04 (1.17% ↓)	49.35 (11.83% ↓)	63.76 (8.05% ↓)	5.46 (0.00% →)	50.65 (15.04% ↑)
2	μ BERT-CodeBERT-LEAM	75.26	91.11	55.97	69.34	5.46	44.03
3	LEAM-CodeBERT-BUGFARM	69.42 (10.91% ↓)	94.11 (1.71% ↓)	41.43 (29.09% ↓)	57.53 (20.72% ↓)	2.59 (0.00% →)	58.57 (40.89% ↑)
4	LEAM-CodeBERT- μ BERT	77.92	95.75	58.43	72.57	2.59	41.57
5	μ BERT-CodeT5-BUGFARM	75.53 (0.72% ↓)	87.7 (0.23% ↓)	59.39 (1.80% ↓)	70.82 (1.17% ↓)	8.33 (0.00% →)	40.61 (2.76% ↑)
6	μ BERT-CodeT5-LEAM	76.08	87.9	60.48	71.66	8.33	39.52
7	LEAM-CodeT5-BUGFARM	71.37 (8.97% ↓)	89.22 (2.42% ↓)	48.6 (22.44% ↓)	62.92 (15.38% ↓)	5.87 (0.00% →)	51.4 (37.65% ↑)
8	LEAM-CodeT5- μ BERT	78.4	91.43	62.66	74.36	5.87	37.34
9	μ BERT-NATGEN-BUGFARM	74.4 (1.37% ↓)	85.01 (0.50% ↓)	59.25 (3.34% ↓)	69.83 (2.17% ↓)	10.44 (0.00% →)	40.75 (5.30% ↑)
10	μ BERT-NATGEN-LEAM	75.43	85.44	61.3	71.38	10.44	38.7
11	LEAM-NATGEN-BUGFARM	69.45 (9.48% ↓)	88.31 (2.86% ↓)	44.85 (24.48% ↓)	59.49 (17.19% ↓)	5.94 (0.00% →)	55.15 (35.80% ↑)
12	LEAM-NATGEN- μ BERT	76.72	90.91	59.39	71.84	5.94	40.61
13	REAL-CodeBERT-BUGFARM	52.97 (4.07% ↓)	55.15 (5.57% ↓)	31.81 (12.39% ↓)	40.35 (9.89% ↓)	25.87 (0.00% →)	68.19 (7.07% ↑)
14	REAL-CodeBERT- μ BERT	55.22	58.4	36.31	44.78	25.87	63.69
15	REAL-CodeBERT-LEAM	57.68	61.44	41.23	49.35	25.87	58.77
16	REAL-CodeT5-BUGFARM	48.91 (5.03% ↓)	48.8 (5.37% ↓)	44.23 (10.50% ↓)	46.4 (8.06% ↓)	46.42 (0.00% →)	55.77 (10.26% ↑)
17	REAL-CodeT5- μ BERT	51.5	51.57	49.42	50.47	46.42	50.58
18	REAL-CodeT5-LEAM	53.79	53.77	53.99	53.88	46.42	46.01
19	REAL-NATGEN-BUGFARM	50.72 (0.26% ↓)	50.6 (0.22% ↓)	60.89 (0.44% ↓)	55.27 (0.32% ↓)	59.45 (0.00% →)	39.11 (0.70% ↑)
20	REAL-NATGEN- μ BERT	50.85	50.71	61.16	55.45	59.45	38.84
21	REAL-NATGEN-LEAM	53.82	53.02	67.1	59.23	59.45	32.9

for the other metrics (average margin of Accuracy=5.97%, Precision=1.48%, Recall=15.5%, and F1-score=10.78%). Furthermore, by looking at the F1 score values, we observe that the models fine-tuned on LEAM have a harder time detecting BUGFARM bugs than those fine-tuned on μ BERT. We believe this is because μ BERT 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 in their dataset, scrapped from the GitHub issue trackers.

5.3.2 Fine-tuning on real-world bugs. To avoid any bias in our conclusion based on synthetic bugs, we also fine-tuned baseline pre-trained models with real-world bugs from three datasets, namely Defects4J [34]⁶, BUGSWARM [60], and RegMiner [57]. The real-world evaluation dataset consists of 723, 3285, and 36412 original and buggy methods from Defects4J, BUGSWARM, and RegMiner, respectively. This resulted in three bug prediction models, i.e., REAL-CodeBERT, REAL-CodeT5, and REAL-NATGEN. We evaluated each model on BUGFARM, μ BERT, and LEAM bugs. The rows 13–21 in Table 2 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 (54.35% on average) and lower Accuracy (50.86% on average), Precision (51.52% on average), Recall (45.64% on average), and F1 score (47.34% on average) values.**

We can also see 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 μ BERT, the bug generation objective does not include any similarity to real-world bugs.

⁶We could use the bugs from Mockito and Closure projects, which were excluded from our subjects since BUGFARM only supports Maven at this time

Table 3. Effectiveness of FitRepair in repairing synthetic bugs. **SI**: Statements involved.

	LEAM [59]	μ BERT [36]	BUGFARM-CodeBERT	BUGFARM-CodeT5	BUGFARM-NatGEN
Total Bugs (SI=1,SI=2,SI>2)	100 (84,15,1)	100 (92,8,0)	100 (52,26,22)	100 (34,25,41)	100 (50,16,34)
Success Rate	34%	49%	27%	22%	23%

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.

5.4 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, μ BERT, and LEAM bugs. The default configuration of FitRepair aims to generate 5000 patches for a single bug and will terminate if this takes more than five hours. Given the size of our subject bugs and the non-negligible patch validation time, we reduced the maximum number of patches to 100^7 (confirmed by the developers of FitRepair that this does not result in drastic performance degradation) and evaluated it on 100 bugs sampled from each bug dataset. When sampling, we controlled for selecting bugs from the same methods for all approaches. To select the methods, we sorted them based on the descending order of method size—measured by the number of characters—and picked the top 100. 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 §2). The collected bugs are from 11 out of 15 subject projects.

The first row in Table 3 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 μ BERT 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 500 bugs took over 50 hours. The second row of Table 3 shows the percentages of bugs that FitRepair successfully patched.

FitRepair successfully repaired 34% and 49% of the LEAM and μ BERT bugs, respectively. In contrast, it can only repair 27%, 22%, and 23% of BUGFARM bugs generated for each pre-trained baseline. This is not surprising, as APR techniques are known to perform better in repairing bugs with SI=1. In fact, 88% of correct patches for μ BERT and LEAM only differ with the corresponding bug in one line. This value is only 64% 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 μ BERT 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 μ BERT 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 \geq 2, we observed the same pattern, i.e., there was at least one statement changing the conditional statements.

⁷This is the max allowance, and the tool often generates a lower number of patches

Automated Bug Generation in the era of Large Language Models

```

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.decoratedBag().add(transformer.transform(value));
        }
    }
    return dec;
}

```

```

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 Object value : values) {
            dec.decorated().add(transformer.transform(value));
        }
    }
    return dec;
}

```

(a) Incorrect method call

(b) Incorrect loop variable type

```

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()) {
        // Removed the array initialization
        bag.clear();
        for (final E value : values) {
            dec.decorated().add(transformer.transform(value));
        }
    }
    return dec;
}

```

```

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) {
            // Removed the method call within the loop
        }
    }
    return dec;
}

```

(c) Removed array initialization

(d) Removed method call

```

public static <E> List<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;
}

```

```

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 Object value : values) {
            dec.decoratedBag().add(transformer.transform(value));
        }
    }
    return dec;
}

```

(e) Incorrect return type

(f) Combination of bug 3a and 3b

Fig. 3. The bugs generated without prompt engineering for the code in Figure 1a.

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.

5.5 RQ4: Necessity of Prompt Engineering

The quality of prompts significantly impacts the quality of the model's response [39]; the more context you provide to the model in the prompt, the better the response it produces. In the context of our problem, identifying the location of Least Attended Statements (LASs) brings three benefits: (1) generating bugs with a similar representation compared to the original code (hard to detect bug), (2) enforcing the model to change multiple lines (hard to repair bug), and (3) reducing the search space, i.e., the location in which the bug can be injected. This will help the model focus on the most important part of the code and generate fewer yet more effective bugs.

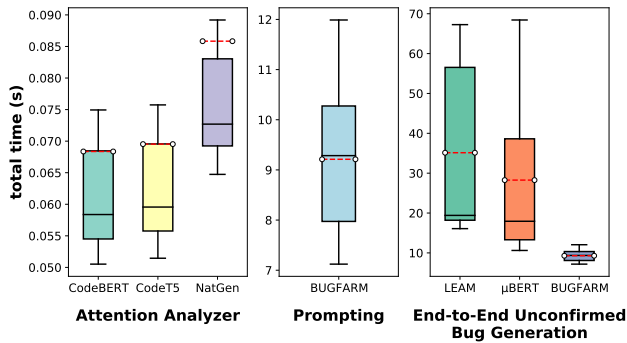


Fig. 4. Performance of BUGFARM compared to alternative approaches in unconfirmed bug generation

Our previous research questions confirm the ability of prompt engineering to generate complex bugs. Generating those bugs without prompt engineering will require more time and cost more. For example, consider prompting an LLM to generate a bug for the code illustrated in Figure 1a, implementing a logic through 13 lines of code (some lines are split for better presentation). There are two least attended statements (LASs) in this method, highlighted in green. The probability that LLM changes one of them is $\frac{2}{13} = 0.15$, i.e., with a probability of 15%, LLM can generate a hard-to-detect bug. However, to ensure the bug is also hard to repair, LLM should ideally modify the two highlighted lines. The probability of LLM choosing the two statements highlighted in green (the LASs) is only 1%, $\frac{1}{\binom{13}{2}} = 0.01$. The code snippets in Figure 3 show different bugs generated by prompting GPT-3.5-turbo six times (buggy lines highlighted in red). We used a simple, not crafted prompt, asking it to generate a compilable bug from the given method. **We can see that the LLM is quite creative in injecting different bugs (statement deletion, changing types, and incorrect method calls) into the code. But none of the responses change any of the least attended statements and, hence, have no overlap with the BUGFARM bugs.**

In general, for a given method with n statements, the bug-inducing change can be applied to any line. BUGFARM selects only k percent of statements to be changed and instructs LLM to modify them, reducing the search space greatly. With the threshold value $k = 20\%$, we can reduce the search space by 80%. Without attention analysis, a naive approach for search space reduction would randomly select k percent of the statements. If not among the least attended statements, they can be easily detected or repaired by learning-based techniques, as shown in previous research questions.

Summary. Prompt crafting help BUGFARM in generating hard-to-detect and hard-to-repair bugs. The chance of generating a similar dataset without prompt engineering is very narrow, if not impossible, due to the relatively large search space for moderate to long methods.

5.6 RQ5: Performance

To assess the performance characteristics of BUGFARM, 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. Looking at *Attention Analyzer* in Figure 4, we can see that 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 *unconfirmed bugs* per method (*End-to-End Unconfirmed Bug Generation* in Figure 4). Compared to attention analysis and prompting, the overhead of other sub-components in BUGFARM, such as parsing and bug selection, is negligible. **Therefore, the average time for generating all the BUGFARM unconfirmed bugs 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 unconfirmed bugs per method.** This is because these approaches generate more unconfirmed bugs, as illustrated in Table 1 under *#UB* columns. For LEAM, this time does not include the training time of the model, which is 24 hours⁸. Also, since μ BERT takes a long time to mutate big classes, we put a timeout of 15 minutes on it to avoid a long generation time. Validation of unconfirmed bugs was also very time-consuming; we spent around 80,000 CPU core-hours to validate over 2.5 million mutants from μ BERT, LEAM, and BUGFARM.

Summary. Compared to alternative approaches, BUGFARM is an efficient and scalable technique for generating bugs. More precisely, BUGFARM is 74% and 67% faster in end-to-end unconfirmed bug generation than LEAM and μ BERT, respectively.

5.7 Discussion

Our comprehensive evaluation of BUGFARM compared to the two most recent alternative approaches confirms the ability of generated bugs to challenge learning-based bug prediction and repair techniques. We want to emphasize that our objective in this paper is to generate hard to detect and repair bugs, concerning learning-based techniques. As a result, comparing with real-world bugs is out of the scope of this paper. We have no claim that BUGFARM bugs mimic real-world bugs (because they do not need to) or that they do not. In fact, BUGFARM leverages GPT-3.5-turbo for generating bugs, which theoretically have seen many real-world bugs in its training data. This can potentially help BUGFARM bugs be similar to real-world bugs compared to μ BERT, which does not concern such similarity. We have identified all the potential threats to the validity in the design of the evaluation and have addressed them throughout the evaluation section when it was the most relevant. Our next step would be investigating the extent to which BUGFARM bugs can enhance downstream applications of synthetic bugs in learning-based techniques, such as test case prioritization and bug localization.

6 RELATED WORK

Our research is related to prior work on (1) real-world bug/vulnerability benchmarks, (2) mutation testing, and (3) learning-based bug/vulnerability generation.

Real-world bug/vulnerability benchmarks. Several attempts have been made to construct real-world bug datasets manually. Defects4J [34], BugSwarm [60], Bugs.jar [53], and RegMiner [57] are the commonly used Java bug datasets that have mined GitHub to collect regression bugs from bug-fixing commits. BigVul [21] and CVEFixes [7] are real-world examples of vulnerabilities collected from bug-fixing reports in the CVE/NVD database [10]. BUGFARM complements the bugs and vulnerabilities in these datasets with hard-to-repair and hard-to-detect bugs. Also, the bugs

⁸This number is quoted from the paper.

in these datasets only represent human mistakes, which could be potentially different from the mistakes AI programming tools make.

Mutation testing. Mutation testing has been widely used in testing programs written in different languages [8, 16–19, 33, 41, 44, 47, 55], as well as testing program properties such as specifications [42], memory usage [65], and energy consumption [31]. Syntactic mutation operators, where small syntactic changes in the code generate the artificial bugs, are not representative of real bugs [35]. Constructing defect-based mutation operators alleviates the mentioned limitation but requires domain knowledge. If done manually, it also can be labor-intensive and time-consuming. BUGFARM is an automated technique for generating bugs; hence, it is superior to non-automated mutation testing. The current implementation of BUGFARM does not generate specific categories of bugs. However, by incorporating some domain knowledge as additional contexts to our prompts, one can expand the bug injection component of BUGFARM to generate specific categories of bugs.

Learning-based bug/vulnerability generation. Learning-based bug generation techniques were first proposed to overcome the limitations of manual defect model construction in mutation testing [11]. Such techniques leverage machine learning to learn the bug or vulnerability patterns from real-world bug fixes and generate mutants accordingly. DeepMutation [61] is such a technique that relies on sequence-to-sequence neural machine translation for learning and generating bugs. SemSeed [50] is a technique that 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 [6] mines bug patterns from historical changes and transforms them into mutation operators semi-automatically.

VULGEN [46] combines pattern mining and deep learning to generate realistic bugs. Specifically, it mines a large corpus of vulnerability-fixing commits to extract bug patterns, and trains a deep learning model on the same dataset to learn where to inject the bugs. LEAM [59] learns to mutate code from large examples of real-world bug-fixing commits. To generate syntactically correct bugs, LEAM represents code as a sequence of AST nodes and learns to apply grammar rules to select and modify the code. Finally, μ BERT [36] produces buggy versions by replacing code tokens with the spacial <mask> token, and uses CodeBERT to predict the masked token. Both LEAM and μ BERT incorporate beam search [23] to generate bugs that involve more than one statement.

BUGFARM is superior to prior learning-based bug-generation techniques in several ways. First and foremost, 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.

7 CONCLUDING REMARKS

Bug benchmarks are essential in software engineering to evaluate automated techniques concerning bugs. The advent of learning-based software engineering demands even more for automated bug-generation techniques, not only for evaluation but also for training or fine-tuning ML4SE models. In this paper, we presented BUGFARM, a model-in-the-loop technique for the automated generation of hard-to-detect and hard-to-repair bugs. Our empirical evaluation shows the superiority of BUGFARM compared to alternative bug generation approaches in (1) generating unique and high-quality bugs, (2) constructing bug benchmarks that are complex both in terms of detection and repair, and (3)

injecting hard-to-detect bugs into arbitrary code. BUGFARM does not rely on existing bug datasets and is model- and programming language-agnostic.

We believe our novel perspective on the bug generation problem offers several directions for future work. First of all, the focus of this paper is on *attention analysis* to identify the weak spots of transformer-based models. While state-of-the-art code-language models and LLMs are all based on transformers, we currently consider alternative approaches generalized to other neural architectures. Second, we plan to evaluate the quality of our bugs for improving other software engineering tasks, including bug localization, test oracles, and test-suite management. Based on the promising results in bug detection and repair, we are confident that our generated bugs are useful for better evaluation and challenging those tasks.

REFERENCES

- [1] 2023. BigQuery Dataset. <https://console.cloud.google.com/marketplace/details/github/github-repos>
- [2] 2023. ManySStuBs4J Dataset. <https://github.com/mast-group/mineSStuBs>
- [3] Open AI. 2023. Open AI ChatGPT. <https://openai.com/blog/chatgpt>
- [4] Amit Seal Ami, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2023. "False negative—that one is going to kill you": Understanding Industry Perspectives of Static Analysis based Security Testing. *arXiv preprint arXiv:2307.16325* (2023).
- [5] Anonymous authors. 2023. Anonymous repository. <https://github.com/projectinvestigator/BUGFARM>
- [6] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. 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)*. IEEE, 268–277.
- [7] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. 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*. 30–39.
- [8] Paul E Black. 2017. Sard: a software assurance reference dataset. (2017).
- [9] Marcel Böhme and Abhik Roychoudhury. 2014. Corebench: Studying complexity of regression errors. In *Proceedings of the 2014 international symposium on software testing and analysis*. 105–115.
- [10] Harold Booth, Doug Rike, and Gregory A Witte. 2013. The national vulnerability database (nvd): Overview. (2013).
- [11] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. 2017. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 511–522.
- [12] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. 2023. Sparks of Artificial General Intelligence: Early experiments with GPT-4. *arXiv preprint arXiv:2303.12712* (2023).
- [13] BugsInPy. 2023. BugsInPy: Dataset of Real-world Python Bugs. <https://github.com/soarsmu/BugsInPy>
- [14] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T Devanbu, and Baishakhi Ray. 2022. 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*. 18–30.
- [15] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [16] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th international symposium on software testing and analysis*. 449–452.
- [17] Marcio Eduardo Delamaro, JC Maidonado, and Aditya P. Mathur. 2001. Interface mutation: An approach for integration testing. *IEEE transactions on software engineering* 27, 3 (2001), 228–247.
- [18] Lin Deng, Nariman Mirzaei, Paul Ammann, and Jeff Offutt. 2015. Towards mutation analysis of android apps. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–10.
- [19] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. 2017. Mutation operators for testing Android apps. *Information and Software Technology* 81 (2017), 154–168.
- [20] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [21] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

- [22] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).
- [23] Markus Freitag and Yaser Al-Onaizan. 2017. Beam search strategies for neural machine translation. *arXiv preprint arXiv:1702.01806* (2017).
- [24] GitHub. 2023. GitHub Copilot. <https://github.com/features/copilot>
- [25] Google. 2023. Google Bard. <https://bard.google.com/>
- [26] Google. 2023. Google PaLM. <https://ai.googleblog.com/2022/04/pathways-language-model-palm-scaling-to.html>
- [27] José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari Sahraoui. 2022. AST-Probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–11.
- [28] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.
- [29] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).
- [30] Ali Reza Ibrahimzada, Yigit Varli, Dilara Tekinoglu, and Reyhaneh Jabbarvand. 2022. 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*. 70–81.
- [31] Reyhaneh Jabbarvand and Sam Malek. 2017. μ droid: an energy-aware mutation testing framework for android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 208–219.
- [32] Yue Jia and Mark Harman. 2009. Higher order mutation testing. *Information and Software Technology* 51, 10 (2009), 1379–1393.
- [33] René Just. 2014. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *Proceedings of the 2014 international symposium on software testing and analysis*. 433–436.
- [34] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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*. 437–440.
- [35] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.
- [36] Ahmed Khanfir, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. 2023. Efficient Mutation Testing via Pre-Trained Language Models. *arXiv preprint arXiv:2301.03543* (2023).
- [37] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.
- [38] Xiangyu Li, Shaowei Zhu, Marcelo d’Amorim, and Alessandro Orso. 2018. Enlightened debugging. In *Proceedings of the 40th International Conference on Software Engineering*. 82–92.
- [39] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *Comput. Surveys* 55, 9 (2023), 1–35.
- [40] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5. Chicago, Illinois.
- [41] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. 2005. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability* 15, 2 (2005), 97–133.
- [42] Evan Martin and Tao Xie. 2007. A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*. 667–676.
- [43] Meta. 2023. Meta LLaMA. <https://ai.facebook.com/blog/large-language-model-llama-meta-ai/>
- [44] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2013. Efficient JavaScript mutation testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 74–83.
- [45] Tai Nguyen and Eric Wong. 2023. In-context Example Selection with Influences. *arXiv preprint arXiv:2302.11042* (2023).
- [46] Yu Nong, Yuzhe Ou, Michael Pradel, Feng Chen, and Haipeng Cai. 2023. Efficient Mutation Testing via Pre-Trained Language Models. *45th IEEE/ACM International Conference on Software Engineering* (2023).
- [47] Vadim Okun, Aurelien Delaire, Paul E Black, et al. 2013. Report on the static analysis tool exposition (sate) iv. *NIST Special Publication* 500 (2013), 297.
- [48] OpenAI. 2023. GPT-4 Technical Report. <https://arxiv.org/abs/2303.08774>
- [49] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2023. Understanding the Effectiveness of Large

- Language Models in Code Translation. *arXiv preprint arXiv:2308.03109* (2023).
- [50] Jibesh Patra and Michael Pradel. 2021. 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*. 906–918.
- [51] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.
- [52] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297* (2020).
- [53] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. 2018. Bugs. jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories*. 10–13.
- [54] Ravi K Samala, Heang-Ping Chan, Lubomir Hadjiiski, and Sathvik Koneru. 2020. 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*, Vol. 11314. SPIE, 279–284.
- [55] David Schuler and Andreas Zeller. 2009. Javalanche: Efficient mutation testing for Java. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 297–298.
- [56] Ensheng Shi, Yanlin Wang, Hongyu Zhang, Lun Du, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Towards Efficient Fine-tuning of Pre-trained Code Models: An Experimental Study and Beyond. *arXiv preprint arXiv:2304.05216* (2023).
- [57] Xuezhi Song, Yun Lin, Siang Hwee Ng, Yijian Wu, Xin Peng, Jin Song Dong, and Hong Mei. 2022. 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*. 314–326.
- [58] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. 2023. Stanford Alpaca: An Instruction-following LLaMA model. https://github.com/tatsu-lab/stanford_alpaca.
- [59] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2022. Learning to Construct Better Mutation Faults. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.
- [60] David A Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 339–349.
- [61] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning how to mutate source code from bug-fixes. In *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 301–312.
- [62] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).
- [63] Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*. 2377–2388.
- [64] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).
- [65] Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, and Jens Krinke. 2017. Memory mutation testing. *Information and Software Technology* 81 (2017), 97–111.
- [66] Chunqiu Steven Xia, Yifeng Ding, and Lingming Zhang. 2023. Revisiting the Plastic Surgery Hypothesis via Large Language Models. *arXiv preprint arXiv:2303.10494* (2023).
- [67] Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. 1–10.
- [68] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 17–26.
- [69] Chaoning Zhang, Chenshuang Zhang, Sheng Zheng, Yu Qiao, Chenghao Li, Mengchun Zhang, Sumit Kumar Dam, Chu Myaet Thwal, Ye Lin Tun, Le Luang Huy, et al. 2023. A Complete Survey on Generative AI (AIGC): Is ChatGPT from GPT-4 to GPT-5 All You Need? *arXiv preprint arXiv:2303.11717* (2023).