Enriching mutation testing with innovative method invocation mutation: Filling the crucial missing piece of the puzzle

Peng Zhang, Zeyu Lu, Yang Wang, Yibiao Yang, Yuming Zhou, Mike Papadakis

***Abstract*—Mutation testing aims to simulate real-world defects, but existing tools often struggle to replicate method invocation defects accurately. To address this, we propose MIN (Method INvocation mutator), which uses a mapping strategy to pair method names with corresponding values, ensuring that methods share argument and return types. This method enhances the feasibility and realism of mutants by considering factors such as library methods, access control, inheritance, and static methods. Experimental results show that integrating MIN into Major (a popular mutation tool) improves semantic similarity to real defects by 11%, increases mutant set diversity to 97.5%, and reduces undetected faults by 38.5%. Furthermore, MIN's performance rivals that of state-of-the-art machine learning-based mutators like CodeBERT, with a 10x speed advantage over CodeBERT and 4x over DeepMutation in generating compilable mutants. These findings demonstrate that MIN can significantly enhance defect simulation and improve the efficiency of mutation testing.**

***Index Terms*—Defects, method invocation, mutator, mutation testing.**

I. Introduction

HIGH-QUALITY defect datasets are of paramount importance in advancing various activities, such as defect prediction, localization, and automatic program repair [1]–[6]. However, a significant challenge faced in this field is the scarcity of large-scale defect datasets, impeding progress in these research domains. To address this limitation, researchers often turn to generating simulated defects using two common methods: traditional mutation mutators and machine learning-based mutators trained with state-of-the-art models [7]–[9]. These approaches help compensate for the limited availability of real-world defect data and facilitate advancements in software engineering research.

A significant proportion of real defects in software can be directly attributed to method invocations, such as calling incorrect methods or passing incorrect parameters. Osman et al. conducted a study based on 717 open-source projects and found that changes to method calls were the most frequent type of edit made in bug-fix code [10]. To further investigate the prevalence of defects related to method calls, we analyzed the Defects4J dataset, focusing on 707 defects, each localized to a single class. Out of these defects, 438 were fixed by updating the code. Since most mutators are designed for updating or replacement, we manually classified these 438 defects. Figure 1 illustrates the results, revealing that more than one-third of the defects were found to be related to method calls.

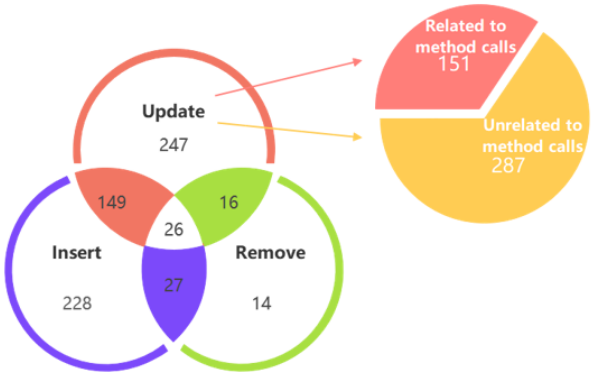

Fig. 1. Defect types in Defects4J for updating AST nodes.

Given the above facts, it becomes evident that mutation testing tools must be capable of simulating method invocation defects. However, existing tools, both traditional and machine learning-based, are inadequate in this area. Traditional mutation analysis tools, for instance, struggle with generating highly targeted mutants for method calls. A case in point is the limitation of the Major mutation tool [12], where only one out of the eight mutators can be directly applied to method calls. Even then, its mutation is restricted to deleting a void method call, which is far from sufficient for simulating the variety of method call defects that might arise.

Learning-based mutators, on the other hand, attempt to predict method names like "`renderWrappedTextBlock`", but they encounter difficulties due to the unique nature of such identifiers. Most pre-trained word embeddings, which are based on natural language, struggle to predict these domain-specific method names accurately. Furthermore, learning-based approaches suffer from significant practical drawbacks. These methods involve encoding the source code into vector inputs and then decoding these vectors back into code, a process that incurs substantial time overhead. This adds to the already burdensome time expense of mutation testing.

Peng Zhang and Mike Papadakis are with Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg. (e-mail: peng.zhang@uni.lu; michail.papadakis@uni.lu)

Zeyu Lu, Yang Wang, Yibiao Yang and Yuming Zhou are with the National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China. (e-mails: zeyulu@smail.nju.edu.cn; njuwy@smail.nju.edu.cn; yangyibiao@nju.edu.cn; zhouyuming@nju.edu.cn)

Corresponding authors: Mike Papadakis and Yuming Zhou (e-mails: michail.papadakis@uni.lu; zhouyuming@nju.edu.cn)

As a result, both traditional and learning-based mutators fail to effectively handle method invocation defects, highlighting a critical gap in current mutation testing tools. There is a pressing need for improved tools that can more effectively address these challenges.

To address this challenge, we introduce MIN (**M**ethod **IN**vocation mutator), a simple and efficient mutator. MIN aims to rapidly identify suitable candidates for method name mutations across the entire project with libraries, replacing the original names. The main requirement for a candidate is to possess identical argument types and return type as the original method, ensuring successful compilation. To this end, MIN follows a three-step process. First, the entire project, including libraries, is parsed to extract the argument types and return type of each method. Second, a mapping is created for each class, associating feasible method names, parameter lists, and control modifiers with the same key values. Third, during mutant generation, we first retrieve the mapped values to obtain the feasible methods. Then we check the control modifiers and modify the argument order to ensure the generated mutant conforms to syntactic rules.

Our experiments demonstrate that integrating MIN into Major yields substantial improvements in performance. Specifically, it yields a 12% enhancement in semantic similarity to real defects and a 6% increase in defect diversity. Moreover, MIN effectively identifies 38.5% of previously undetected faults, underscoring the limitations of existing mutation operators in replicating MIN's capabilities. Additionally, these mutants are subsuming mutants. Please refer to RQ4 for a detailed explanation. Notably, MIN outperforms CodeBERT [1] in terms of efficiency. The compilation pass rate of MIN is 6.4 times higher than that of CodeBERT, making it more conducive to integration into Major. In summary, MIN contributes a valuable addition to Major's functionality, effectively filling a crucial gap in the mutation testing puzzle. Its simplicity, efficiency, and effectiveness make it a compelling choice for enhancing mutation testing in software development projects.

In summary, our work makes the following key contributions:

- In our study on mutant generation, we demonstrate that practicality does not necessarily align with complexity. Despite the allure of complex methods, a traditional mutation testing tool, with enhancements addressing its weaknesses, can outshine intricate approaches. Thus, we introduce a novel mutator named MIN, designed to address the critical gap of lacking a method invocation mutation operator in existing mutation testing tools.
- We undertake a comprehensive experiment to assess the efficacy of MIN from multiple perspectives, including semantic similarity, diversity of mutant sets, efficiency, and fault detection capabilities. The experimental findings demonstrate that MIN significantly enhances the overall effectiveness of mutation testing tools.

[1]In this study, CodeBERT is used specifically as a masked language model (MLM) for generating mutants by predicting method names and arguments. Unlike general-purpose applications of CodeBERT, our approach focuses solely on mutations related to method invocations. For clarity and brevity, we refer to this specific usage as "CodeBERT" throughout the paper.

```java
//Original Code
public void printWrapped(PrintWriter pw, int width, int nextLineTabStop, String text)
{
        StringBuffer sb = new StringBuffer(text.length());

        renderWrappedTextBlock(sb, width, nextLineTabStop, text);
        pw.println(sb.toString());

}


//Real Defect
public void printWrapped(PrintWriter pw, int width, int nextLineTabStop, String text)
{
        StringBuffer sb = new StringBuffer(text.length());

        renderWrappedText(sb, width, nextLineTabStop, text);
        pw.println(sb.toString());

}
```

Fig. 2. A defect indexed as Cli-33 in Defects4J.

- We have made our MIN implementation and experiment openly accessible [33], enabling other researchers and practitioners to replicate and build upon our findings.

The other sections of this paper are arranged as follows: Section II presents a motivating example. Section III elucidates the specific methodology of MIN. Section IV explains the experimental design. Section V reports in detail the experimental results. Section VI introduces relevant work. Section VII provides some interesting discussions on MIN. Section VIII addresses potential threats in the experiment. Section IX concludes with a summary of the entire paper.

II. MOTIVATING EXAMPLE

Figure 2 illustrates the defect in Defects4J [11], [13] indexed as Cli-33. In this particular case, the method `renderWrappedText` is mistakenly called instead of `renderWrappedTextBlock`[2]. Now, let us consider simulating this defect.

In the context of traditional mutation operators, MuJava [14] does not provide a mutator applicable to this line of code. However, in PIT [15], [16] or Major [12], there is a mutator called "void method calls" that can be used for this purpose. It works by removing a void method call, effectively deleting the corresponding line. This mutation resulted in 17 failing test cases, but only one of them exposed the real defect. To quantitatively assess the semantic similarity between the mutant and the actual defect, we employed the Ochiai coefficient, which was calculated to be 0.243 (Ochiai = $1/17^{0.5}$, as explained in Section IV). The coefficient indicates a low correlation between the mutant and the real defect.

It is essential to mention that in addition to the "void method calls" mutator in PIT, two more mutators involving method calls are available [17]. The first mutator varies the return value of a non-void method to one of (false, 0, 0.0, 'u0000', null), while the second mutator takes one of the arguments of a non-void method as the return value directly. However, these mutators are not applicable to the specific void

[2]For Cli-33, there is no method `renderWrappedTextBlock` in the buggy version. It is added to fix the bug. However, our purpose is to simulate the bug. As the result, replace the method call is enough and we do not need to delete the method `renderWrappedTextBlock`.

method in Figure 2. This observation highlights the limited diversity of traditional mutators for method calls, which makes it challenging to generate adaptive mutants tailored to the unique characteristics of each project. Furthermore, it is worth noting that mutators involving method calls constitute only a small proportion of the total mutators available in the three tools (MuJava: 0/15, Major: 1/8, PIT: 3/29). This ratio does not align with the occurrence of this type of defect among real defects, indicating that the existing set of mutators is not adequately representative of real-world scenarios.

Let us examine the learning-based mutation operators that utilize CodeBERT [25] as a model to generate mutants [8], [21]. Initially, the CodeBERT-based mutation operator masks the code as follows:

```
StringBuffer sb = new StringBuffer(text.length
    ());
<mask>(sb, width, nextLineTabStop, text);
pw.println(sb.toString());
```

Afterward, the CodeBERT model predicts that the top five likely tokens to replace the masked one are `print`, `write`, `format`, `wrap`, and `append`. Unfortunately, none of these five mutants can be successfully compiled. The reason behind this unsatisfactory outcome is the uniqueness of the identifier `renderWrappedText`, which is likely scarcely represented in the training set. Since CodeBERT is already a code-based pre-trained language model, there is no reason to believe that it will achieve better results when using a pre-trained model based on natural language (e.g., BERT [18]). In the following, for brevity, we will use CodeBERT to refer to this mutation generation approach.

Although SemSeed [7], another learning-based approach, attempts to address the issue of unbound tokens (e.g., "`renderWrappedTextBlock`"), its prediction capability is limited to the current file and the 1000 most frequent tokens. This limitation highlights a general drawback of learning-based approaches: when the expected token is rarely encountered in the training data, relying on word embeddings to obtain the correct answer becomes highly improbable. Notably, the uniqueness of method invocation identifiers poses a significant challenge for learning-based approaches.

As illustrated in the example above, existing traditional mutators prioritize generality, lacking the necessary diversity and effectiveness in handling method invocations. Learning-based mutators also encounter challenges, particularly in unique identifier cases. However, method invocation defects are prevalent and frequently encountered in practical scenarios. As a result, there is a compelling need for the development of a straightforward and flexible method invocation mutator that can adapt to any project. Such a mutator would represent a significant advancement in the field of mutation testing, enhancing defect simulation capability in method invocations.

III. APPROACH

Figure 3 provides a high-level overview of MIN. The core concept behind MIN is to identify feasible methods along with their arguments. During the mutant generation process, feasible methods replace the original method, and the order of arguments may be adjusted to ensure successful compilation. To enhance the effectiveness and relevance of the mutants generated, MIN considers several factors, which are detailed in different sections of the paper. These factors include library methods (Section III.A), access control modifiers (Section III.C), inheritance relationships (Section III.C), and the majority of static methods (Section III.C).

Section III.A explains the procedure of parsing Java projects. Subsequently, Section III.B demonstrates how MIN creates a mapping system for efficient matching of viable methods. Finally, Section III.C describes the process of utilizing the created mapping system to generate mutants effectively.

A. Parsing Java Projects

To explore viable approaches for generating mutants, we undertake a comprehensive parsing of the entire project. For this task, we employ JavaParser [19], [20], a well-regarded parser for the Java language known for its capacity to extract essential information like modifiers, argument types, and return types of each method upon its definition. This enables us to efficiently parse, analyze, transform, and generate code.

In order to ensure effective searching, we categorize these methods based on the class in which they are defined. Additionally, to handle the replacement from library methods, we also parse the Java Development Kit (JDK) classes to gather crucial information on the methods they contain. For other library methods, we can acquire the necessary information by parsing the corresponding source code. It is important to mention that, during the mutation process, we exclude generic methods. This decision is driven by the fact that the determination of their argument types occurs only when they are invoked, making their inclusion in the mutation process impractical.

B. Quick Match by a HashMap

After parsing each class, we retrieve all the methods, including their modifiers, return types, and argument types. This section will detail the process of computing a distinctive value for a set of methods belonging to the same class. To achieve this, we begin by assigning a distinct value for the i-th type A, which is initially present in both the JDK and the project. Specifically, $v(A) = \text{Prime}(i)$, where $\text{Prime}(i)$ represents the i-th prime number.

Next, we categorize all methods into four distinct groups: (1) void methods without arguments, (2) void methods with arguments, (3) non-void methods without arguments, and (4) non-void methods with arguments.

Finally, we calculate a unique value for each method using various formulas:

- Group A: For a void method m without argument: $map(m) = 1$;
- Group B: For a void method m with argument types $(t_1, t_2, \ldots, t_n)$:

$$map(m) = \prod_{j=1}^{n} v(t_j)$$

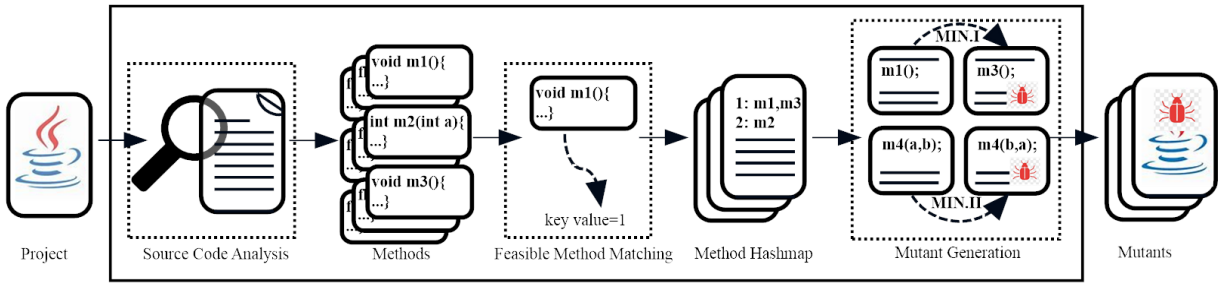

Fig. 3. The process of MIN to generate mutants.

- Group C: For a method m return type t_1 without argument:

$$map(m) = e^{-v(t_1)}$$

- Group D: For a method m return type t_1 with argument types $(t_2, t_3, \ldots, t_n)$:

$$map(m) = e^{-v(t_1)} * \prod_{j=2}^{n} v(t_j)$$

The map values obtained through the aforementioned calculations offer the following advantages:

- The map values for the four groups of methods are unique and different from each other;
- In Group B or Group D, methods with the same set of argument types share the same map value, irrespective of their order of appearance;
- For any given map value within a class, the corresponding set of argument types and the return type can be uniquely determined.

The proofs concerning these properties can be found in the supplementary material.

After computing the map values for each method, we construct a single HashMap to store the methods. Each entry in the HashMap includes the method's corresponding map value, as well as its modifier. The order of argument types is encoded by primes within the map values, as reflected in Table I. It is worth mentioning that the HashMap for JDK is already prebuilt in MIN, streamlining the storage process for JDK methods.

To help understand the real-world distribution of these method types, we analyzed these JDK methods. The results are as follows:

- Group A (void methods without parameters): 745 methods (4.2%)
- Group B (void methods with parameters): 3294 methods (18.6%)
- Group C (methods with return values but no parameters): 6416 methods (36.1%)
- Group D (methods with both return values and parameters): 7285 methods (41.1%)

This distribution offers insights into the prevalence of different method types and informs the mapping method's behavior when applied to real-world subjects.

C. Generating Mutants

To generate mutants using MIN, we modify two aspects of method invocations: (I) the method name and (II) the method parameter list. Regardless of the modification type, the first thing is to identify a list of feasible methods. The specific steps are as follows:

1) For a given method call in the source code within class `ClassA`, we determine the class `ClassB` in which the method is defined.
2) We then search the ancestor classes of `ClassB` to construct a class set `S`.
3) Feasible methods are identified based on the following rules: (a) if `ClassA` is equal to `ClassB`, we search the methods with the same key value (as described in Section III.B) in `S`'s HashMap for "non-private" methods and `ClassB`'s HashMap for all methods; (b) if `ClassA` is a child of `ClassB`, we search the the methods with the same key value in `ClassB`'s and `S`'s HashMap for the "non-private"; and (c) in all other cases, we only search the the methods with the same key value in `ClassB`'s and `S`'s HashMap for "public" and "static"[3] methods.

These rules ensure that the appropriate set of methods is considered for mutant generation, taking into account class relationships and access modifiers. This approach allows for the generation of contextually relevant mutants while preserving the program's structural integrity.

Meanwhile, overloading does not introduce ambiguity, as overloaded methods have distinct argument type lists. Thus, in cases where overloading occurs within a class, the feasible method list for each overloaded method remains unambiguous, since the HashMap key is uniquely determined by the argument type list. This ensures the correct methods are selected for mutation, with consideration of the specific argument types associated with each method call.

Once a list of feasible methods is obtained, the original method call can be mutated in the following three ways:

1) **Mutating only the method name:** This requires the parameter type lists of the original and alternative methods to be identical. Even if there are multiple parameters

[3]In this paper, in Section III.A, for "static" methods, we parse it only when it is "public". So, we consider static as a kind of access control modifier. When we want to replace a static method, only the static methods will be searched. Meanwhile, when we want to replace a non-static method, the static methods will be put into the candidate.

TABLE I
HASHMAP EXAMPLE FOR CLI-33.

	Method 1 in the Value List			Method 2 in the Value List			...
Key	name	arg.	modifier	name	arg.	modifier	...
2014	appendOption	[53,2,19]	public	...	...	...	...
3e+214	getArgName	[]	public	getNewLine	[]	public	...
...	...	...	...	...	...	...	...
4e+199	renderWrappedTextBlock	[53,59,59,5]	public	renderWrappedText	[53,59,59,5]	public	...
...	...	...	...	...	...	...	...

of the same type, the parameter list remains unchanged in this scenario. For example, consider the methods `void MethodA(int a, float m, int b, int c)` and `void MethodB(int x, float n, int y, int z)`. Then `MethodA(a, m, b, c)` would be modified to `MethodB(a, m, b, c)`.

2) **Mutating only the method parameters:** This scenario can be divided into two cases:
   1. **When the parameter list contains at least two arguments of the same type:** In this case, all arguments of the same type are reordered in reverse order. For example, consider the method `void MethodA(float n, float m, int b, int c)` . Then `MethodA(n, m, b, c)` would be modified to `MethodA(m, n, c, b)`.
   2. **When the parameter list contains no arguments of the same type :** If there exists another method implementation with a different parameter order, MIN can generate mutants by invoking that specific polymorphic method with the corresponding arguments. For example, consider the methods, `void MethodA(float n, int c)` and `void MethodA(int c, float n)` Then `MethodA(n, c)` would be modified to `MethodA(c, n)`.
3) **Mutating both the method name and the method parameters:** Consider the methods `void MethodA(int a, float m, int b, int c)` and `void MethodB(float n, int x, int y, int z)`. These methods can be interchanged as they share the same "map value". However, to align with the alternative method's parameter order, adjustments are required: we will reverse the arguments with the same type. For example, the three `int` parameters are reordered in reverse order, resulting in a mutation from `MethodA(a, m, b, c)` to `MethodB(m, c, b, a)`. Note that by the "reverse order", we only generate one mutant. This is to avoid combinatorial explosion caused by exhausting all the order.

Besides, we limit MIN to generate at most 10 mutants for a method call. This can be adjusted in the code according to actual needs.

D. Example for Cli-33

To demonstrate the process of generating mutants on method invocation, we use the Cli-33 project as an example. First, parsing the entire Cli project yields 543 methods. Second, we construct a dictionary for all the types that appear in the project. The dictionary contains information about the prime numbers for each type: {'Option': 2, 'OptionGroup': 3, 'String': 5,..., 'URL': 109}. Next, we build a HashMap for each class in the Cli project. Because the bug is in `HelpFormatter.java`, and the buggy method is also defined there, we only need to focus on the HashMap built by `HelpFormatter.java` in this example. The corresponding HashMap is in Table I.

Then when the code contains the call `renderWrappedTextBlock(sb, width, nextLineTabStop, text)`, we aim to generate mutants for this method. We calculate the key `Key = 4e+199` based on `renderWrappedTextBlock`. Using this key, we query the `HashMap` (shown in Table I) to retrieve two feasible methods: `renderWrappedTextBlock` and `renderWrappedText`. These methods are identified as valid candidates for mutation, as they share the necessary characteristics to replace the original method in the mutation process. Then we can check if it can generate 3 types of mutants:

1) **Mutating only the method name:** Since the two parameter lists are the same ([53,59,59,5]), the first mutant $mutant_1$ is: `renderWrappedTextBlock(sb, width, nextLineTabStop, text) ->renderWrappedText(sb, width, nextLineTabStop, text)`
2) **Mutating only the method parameters:** Since there are two "59" in the parameter list, the second mutant $mutant_2$: `renderWrappedTextBlock(sb, width, nextLineTabStop, text) ->renderWrappedTextBlock(sb, nextLineTabStop, width, text)`
3) **Mutating both the method name and the method parameters:** Since there are two "59" in the parameter list, we can reserve them when modifying the method name to get the third mutant $mutant_3$: `renderWrappedTextBlock(sb, width, nextLineTabStop, text) ->renderWrappedText(sb, nextLineTabStop, width, text)`

It is noteworthy that we observe $mutant_1$ being identical to the real fault in semantics, indicating that our mutation process was successful in simulating the fault.

IV. Experimental Design

This section presents the experimental design, including research questions, benchmarks, setup, methodology, and procedure. Our goal is twofold: (1) to address the limited ability of traditional mutation tools to generate method invocation-related mutants, and (2) to assess whether complex, large-model-based approaches offer a cost-effective advantage over our simpler technique.

A. Baselines

Strictly speaking, only CodeBERT and DeepMutation serve as baselines, as our tool is intended to complement existing mutation testing rather than replace it. However, we still present specific results for the full spectrum of mutation operators in Major as a control, allowing readers to compare the quality of individual mutation operators. [4] [5]

1) Major: Major [12] is a widely adopted Java mutation testing tool. In this paper, we employ all available mutators (i.e., AOR, LOR, COR, ROR, SOR, ORU, STD, and LVR) to generate the complete set of possible mutants. However, it is essential to note a crucial detail about Major's behavior. Before conducting mutation testing, Major collects coverage information. Consequently, even for mutants can not be compiled, Major labels them as "killed" if the test cases cover the modified line. In other words, an uncompilable mutant is considered "killed" if the test cases execute the code containing the mutation. However, for MIN, CodeBERT and DeepMutation, we label the uncompilable mutant as "survived". See the Section VII.C for the reasons.

2) CodeBERT: CodeBERT [25] is a pre-trained language model for programing languages. In [8], [21], they used CodeBERT to generate mutants via a Masked Language Modeling (MLM) task. MLM takes a sequence containing a masked token as input and predicts a suitable replacement. In our study, CodeBERT is exclusively used to mutate code related to method calls. For instance, the original code "A(B,C)" is transformed into three masked snippets: "<mask>(B,C)", "A(<mask>,C)", and "A(B,<mask>)". Then, 5 predictions are used to generate mutants for each snippet. At the same time, we will discard the prediction which is the same to the original.

We only take the method name and parameters into consideration because most of the mutants for other token types have already been covered by the Major mutators.

3) DeepMutation: DeepMutation [9] is a tool that leverages deep learning techniques to enhance mutation testing capabilities. Building on recent advancements in machine learning, DeepMutation utilizes a Recurrent Neural Network Encoder-Decoder architecture to learn mutants from a vast dataset comprising approximately 787,000 real faults extracted from software programs. This approach complements existing mutation testing frameworks and enhances their fault-detection capabilities with mutants learned from actual program faults. For our comparison, we utilized the open-source code from the DeepMutation repository without any modifications to generate the mutants.

[4]We opt for Major over PIT for two reasons: 1. PIT still exhibits weak performance in handling method calls (see Section II for a motivating example); 2. By default, most of PIT's operators overlap entirely with those of Major. Therefore, we deemed it satisfactory to utilize Major as a representative of traditional mutation tools.

[5]Our considerations for not using IBIR as a baseline are detailed in "Section VI related work". In short, IBIR does not provide an operator for modeling method call defects.

B. Benchmark

In our comparison of mutators, we leverage Defects4J [13] 2.0.0, a benchmark comprising 17 real-world Java projects. This benchmark includes over 800 reproducible real faults extracted from source code repositories. For each fault, the benchmark provides a buggy version, a fixed version, and at least one triggering test case within a manually written test suite. Note that we only generate mutants on classes (that were) modified by the bug fix, which is the common practice in existing studies [21], [22].

For MIN, we successfully executed the mutator on 460 bugs from the 17 projects, where both Major and MIN mutators executed within 30 minutes on the fixed versions. However, 26 of these bugs were filtered out for MIN as all mutants generated by MIN were uncompilable.

For CodeBERT, we obtained results for 289 bugs, with successful execution within the same time frame. Out of these 289 bugs, 4 were filtered out because all mutants were uncompilable.

For DeepMutation, which supports only six projects (Chart, Closure, Lang, Math, Mockito, and Time), we obtained fewer results. DeepMutation successfully generated mutants for 152 bugs, producing a total of 114,703 mutants. Since DeepMutation generates mutants for all the classes instead of modified classes, we identified 44 bugs that had at least one runnable mutant in the classes (that were) modified by the bug fix.

Finally, of the 44 defects with runnable mutants in DeepMutation, 19 defects had mutants generated by all three approaches: MIN, CodeBERT, and DeepMutation.

C. Research Questions

RQ1 (Simulation capability): How does MIN compare with Major's single mutator, DeepMutation and CodeBERT in simulating defects?

The focus of RQ1 is to compare MIN's ability to simulate defects with that of Major's single mutator, DeepMutation and CodeBERT, while disregarding efficiency. Each mutation operator aims to generate mutants to the best of its ability. For each fault, we calculate the "maximum semantic similarity" and "diversity" metrics (See next subsection for more details).

RQ2 (Gain of integration into Major): What improvement is achieved by integrating DeepMutation, CodeBERT or MIN into Major?

The objective of RQ2 is to investigate the advantages gained from integrating DeepMutation, CodeBERT or MIN into the pre-existing Major tool. It is essential to consider that an operator might be efficient but redundant, as its functionality could already be covered by the current set of operators in Major. In order to answer RQ2, we adopt a simulation approach by combining and analyzing the kill matrices of

each tool, avoiding direct modifications to the Major source code for integration. This allows us to assess the impact of integration and understand the improvements achieved through the combination of these tools.

RQ3 (Efficiency of defect simulation): How does MIN compare to DeepMutation and CodeBERT in terms of the time it takes to generate a mutant?

The objective of RQ3 is to explore the efficiency of different mutation tools. To assess this, we compare the average time taken by MIN, DeepMutation, and CodeBERT to generate a single mutant.

RQ4 (Fault Detection capability with subsumption): Does MIN detect more defects than DeepMutation and CodeBERT?

The objective of RQ4 is to explore the defect detection capability of MIN in comparison to DeepMutation or CodeBERT. To evaluate fault detection capability, we employ two approaches: minimal test suite for killing mutants and subsuming mutants revealing faults. Details of these approaches are provided in the next section.

D. Experimental Setup and Evaluation Methodology

We measure the effectiveness of a mutator from four perspectives: **(1) the quality of individual generated mutant, (2) the quality of the entire set of generated mutants, (3) efficiency, and (4) fault detection capability with subsumption**.

1) Semantic similarity: When generating mutants, semantic similarity is often considered more important than syntactic similarity [21]. To evaluate the semantic similarity between a mutant *m* and a real defect *d*, one commonly used approach is the Ochiai coefficient [21], [23]. The Ochiai coefficient is computed as follows:

$$Ochiai(m,d) = \frac{|fT(m) \cap fT(d)|}{\sqrt{|fT(m)| \times |fT(d)|}} \tag{1}$$

Here, *fT* denotes the set of test cases that fail when executed on program. The Ochiai coefficient quantifies the overlap in failing test cases between a mutant and the real defect, providing insight into how closely the mutant replicates the defect's behavior.

It is important to note that while a mutator may generate many mutants for a given class, each defective class in commonly used defect benchmarks (e.g., Defects4J) is typically associated with a single documented real-world defect. This is because these benchmarks are carefully curated to capture specific bugs found in real-world software. As a result, most mutants may not be specifically designed to simulate that particular defect. Consequently, the maximum value of the Ochiai coefficient among these mutants reflects the mutator's capability to accurately simulate the specific defect.

2) Diversity: Generating diverse mutants is a crucial aspect of an effective mutator. However, quantifying the diversity among mutants can be challenging, especially when most mutants exhibit high syntactic similarity with edit distances of less than 3. Specifically, the syntactic distance [21] is often measured in terms of the number of tokens changed, as well as the Bleu score, which compares the similarity between the generated mutants and the real faults. A smaller edit distance typically implies that the mutated code closely resembles the original code in structure, leading to a high syntactic similarity. To overcome this challenge, we can shift our focus to the concept of diversity from a semantic perspective. Diverse mutants can be understood as a set of mutants that excel at distinguishing between different test suites. For instance, we can consider a highly diverse mutant set capable of generating distinct scores for all non-redundant test suites. As the diversity decreases (e.g., by removing some mutants), certain test suites may transition from having different scores to identical scores. To measure the loss of diversity, we can calculate the ratio of pairs of test suites where the score relationship changes from inequality to equality. This measure, known as Order Preservation (OP) [24], quantifies the preservation of the mutation score order, and we utilize it in our study as a measure of diversity. A higher value of OP indicates a greater level of diversity among the mutants.

To evaluate the diversity of the complete set of generated mutants, which includes Major, MIN, DeepMutation and CodeBERT, we start by assuming that this set exhibits the highest level of diversity, represented by OP=1. Next, we systematically analyze subsets of mutants and examine how well they maintain the mutation score order among test suites compared to the entire set of mutants. This comparison enables us to assess the preservation of the mutation score order and, subsequently, the level of diversity within each subset. By employing this measure, we can gain valuable insights into the diversity of mutants generated by different approaches and effectively assess their capability to produce distinct mutants [6].

To compute OP, we first generate a sequence of test suites:

$$T_0 \supset T_1 \supset T_2 \supset \cdots \supset T_k \neq \varnothing$$

where $|T_{i+1}| = \text{int}(|T_i| \times 0.5)$ for $i = 0, 1, ..., k-1$, and $k = \text{int}(\log_2 |T_0|)$. Next, we compute mutation scores against the complete set (M) and the subset (M_s, e.g., MIN mutants) , resulting in $2(k+1)$ mutation scores. Then, we assess the loss of diversity by counting the number of pairs that have changed mutation scores as follows:

$$X = \left\{ i \mid \begin{array}{l} MS(M, T_{i+1}) < MS(M, T_i) \\ \text{and}, MS(M_s, T_{i+1}) = MS(M_s, T_i) \end{array} \right\}$$

Finally, we determine OP using the formula:

$$OP(s) = 1 - \frac{|X|}{k}$$

To reduce randomness when generating sequences, we repeat the computation of OP for 20 times to obtain an average value.

[6]We aim to elucidate the concept of diversity through an analogy: we can liken each mutant to a judge in a competition, with different sets of test cases representing the competitors. The diversity of mutants entails preserving a range of preferences among these judges. Consequently, if, after reducing the number of judges, competitors who originally received different scores now achieve the same score, we perceive this as a loss of diversity.

3) Efficiency: Efficiency is critical in mutation testing due to the substantial overhead it often imposes. To gauge the practicality of MIN, we compare it with DeepMutation and CodeBERT based on the average time required to generate a single mutant.

For MIN, the reported average time includes the overhead of parsing the entire project, even though mutants are only generated for the classes (that were) modified by the bug fix. Thus, if MIN were used to generate mutants for the entire project, the average time would likely be shorter than reported.

For CodeBERT, mutants are generated only for the modified classes, but as CodeBERT processes fixed-length code snippets for mask prediction, the time remains consistent, whether applied to a single class or the entire project.

For DeepMutation, it parses the entire project and generates mutants for the entire project. Therefore, the reported average time accounts for mutants generated across the entire project.

We did not consider the average execution time of mutants in our analysis due to potential compilation failures, which could distort the results. Instead, we focus on the time required to generate mutants as the primary efficiency metric. We also report the proportion of successfully compiled mutants for each mutation operator as supplementary information. Additionally, the average test execution time for each compilable mutant is included in the supplementary materials, as mature mutation tools use optimizations (e.g., skipping test cases after a mutant is killed) to speed up execution. Since CodeBERT and MIN have not yet implemented such optimizations, comparing their execution times with Major would be unfair, which is why this information is provided separately in the supplementary materials.

4) Fault detection capability with subsumption: An important aspect of assessing the quality of the mutation test is its ability to reveal defects. In this study, the "fault detection capability of mutants" refers to the ability of test cases—designed or selected based on mutants—to detect real faults in the program. Although mutants themselves do not detect faults, they serve as proxies for potential defects, helping to design or identify test cases that can effectively detect faults when executed. We employ two approaches to evaluate fault detection capability:

1) Minimal test suite for killing mutants: Drawing from existing studies [28]–[32], our goal is to discover the smallest test suite capable of eliminating a given set of mutants. Subsequently, we assess whether this test suite can effectively uncover actual defects in the code;
2) Subsuming mutants revealing the faults: We aim to identify how many mutants possess the following characteristics: they reveal defects previously undetected by Major and are subsuming. To achieve this, we identify *undetected faults* where triggering test cases does not contribute to Major's mutation score (i.e., deleting them will not decrease the mutation score). Then, we add the new mutants. If any new mutant is only killed by any triggering test case, we can obtain that a) this mutant is a subsuming mutant; and b) this mutant reveal the fault.

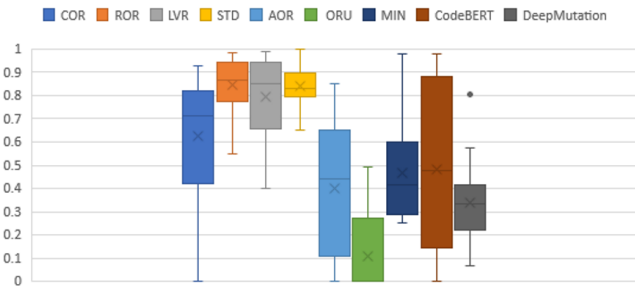

Fig. 4. (RQ1) Diversity for all mutators.

E. Experimental Procedure

To evaluate the mutators, we need to constructe the *kill matrix* (i.e., information on whether a test case can kill a mutant). The procedure is as follows:

- For **Major**, the results can be directly obtained by running the `defects4j mutation` command in Defects4J.
- For **MIN**, **CodeBERT**, and **DeepMutation**, the process includes:
  1) Generating the mutant files.
  2) Replacing the original files with the generated mutant files.
  3) Executing the tests to identify which test cases fail for each mutant.

Finally, all results are aggregated to construct the kill matrix. See more details in [33].

V. RESULTS

In this section, we report in detail the results, including the efficiency-unbounded simulation capability, the gain of integration into Major, the efficiency of defect simulation and the fault detection capability.

A. RQ1: Simulation capability

Table II presents the average semantic similarity for each mutator. Taking MIN as an example, MIN successfully generated 42,619 mutants for 434 faults, including both compilable and non-compilable mutants. Then for each fault, the highest Ochiai coefficient is recorded. Finally, the average value is 0.467 for 434 highest Ochiai coefficient values. The results clearly indicate that MIN performs exceptionally well and ranks among the top mutation operators in Major. Considering its superior performance, integrating MIN into the existing set of mutation operators in Major appears to be a viable option.

Among Major's mutators, STD (STatement Deletion), COR (Conditional Operator Replacement), and MIN demonstrate efficiency by generating a smaller number of mutants per fault ($<$100) while maintaining a higher semantic similarity ($>$0.450). Conversely, the AOR (Arithmetic Operator Replacement) mutator is deemed inefficient as it produces a large number of mutants but with a lower semantic similarity below 0.4. The ORU (Operator Replacement Unary), LOR (Logical Operator Replacement), and SOR (Shift Operator Replacement) mutators have lower frequencies and show weaker defect simulation abilities. Meanwhile, we must admit that

TABLE II
(RQ1) AVERAGE SEMANTIC SIMILARITY FOR EACH MUTATOR.

	COR	ROR	LVR	STD	AOR	ORU	LOR	SOR	CodeBERT	MIN	DeepMutation
#faults	436	456	447	453	326	116	94	70	285	434	44
#mutants	34210	53904	54363	30743	55748	1416	2564	1736	342444	42619	462
#mutants/fault	78.5	118.2	121.6	67.9	171.0	12.2	27.2	24.8	1201.6	98.2	10.5
Avg. Ochiai	0.454	0.469	0.478	0.475	0.383	0.131	0.115	0.100	0.513	0.467	0.132

Note: The number of mutants includes both compilable and non-compilable mutants.

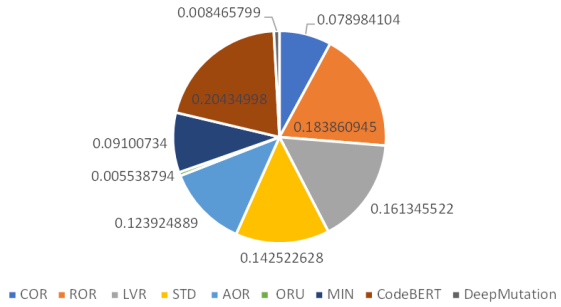

Fig. 5. (RQ1) Percentage of different mutants.

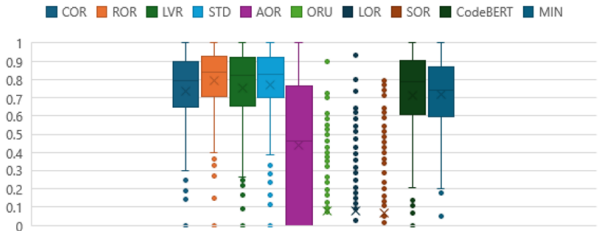

Fig. 6. (RQ1) Excluding DeepMutation to show diversity for more bugs.

ROR (Relational Operator Replacement) and LVR (Literal Value Replacement) demonstrate a higher semantic similarity to defects on average compared to MIN. However, our goal is not to replace traditional mutation operators with MIN. As expressed in the title, "Filling the Crucial Missing Piece of the Puzzle," our intent is to position MIN as a complementary addition to traditional mutation testing tools. As a result, after acknowledging the initial success of MIN, we are more concerned about whether combining MIN with Major can significantly improve the effectiveness of Major.

CodeBERT achieves the highest semantic similarity among the mutators but also generates the most mutants. Meanwhile, DeepMutation achieves the lowest semantic similarity with the least mutants. To explain this, **it is important to note that** semantic similarity may be influenced by the mutant that most closely resembles the real defect, leading to **potential confounding effects due to mutation size**. As a result, concluding that CodeBERT is the best mutator based solely on semantic similarity might not be appropriate (**See more details in Section VII.A**). Overall, MIN stands out as an efficient and effective mutator in terms of semantic similarity, showing promise for integration into the Major mutation operator framework.

Next, we delve into the diversity of defects, a metric that encompasses the value of the entire mutant set rather than individual mutants. To compare MIN, CodeBERT, and DeepMutation, we analyze all 19 bugs for which all three approaches successfully generate mutants. Figure 4 illustrates the diversity across these 19 bugs. Before delving into Figure 4, it's prudent to direct our attention to Figure 5, which showcases the percentage of mutants generated by each operator after summarizing all 19 defects. This step is crucial as the diversity contributed by a subset correlates closely with its proportion of the full set. Therefore, understanding each operator's share of the full set is paramount. We find that MIN accounts for 9%, CodeBERT for 20%, DeepMutation for 0.8%, and Major (comprising COR, ROR, LVR, STD, AOR, and ORU) for 70% (with LOR and SOR generating 0 mutants for the 19 bugs). Then back to Figure 4, across these 19 bugs, MIN achieved comparable results (0.469) to CodeBERT (0.480), with both significantly outperforming DeepMutation (0.339). This indicates that while CodeBERT slightly surpasses MIN in diversity, their performance remains close. In contrast, DeepMutation falls considerably behind. The results highlight the effectiveness of both MIN and CodeBERT in generating a diverse set of mutants, which is crucial for simulating real-world defect variations.

However, due to the small number of defects available for DeepMutation, the overall sample was low, and to further compare the diversity of MIN and CodeBERT, we compared on CodeBERT, the 285 defects for which compilable mutants were generated. Figure 6 shows the comparison results, and we can see that MIN's performance (0.717) is comparable to CodeBERT's (0.733).

Conclusion: RQ1 - Simulation Capability

The analysis highlights MIN as a strong performer in defect simulation, achieving an average Ochiai coefficient of 0.467 across 434 defective classes within the Major framework. This positions MIN among the top operators, showcasing its potential to enhance defect simulation tasks efficiently. CodeBERT, while achieving the highest semantic similarity (0.513 across 285 defects), generates ten times more mutants than other operators. This increase in mutants may contribute to its higher similarity but raises concerns about efficiency and computational cost.

In terms of mutant diversity, MIN performs competitively with CodeBERT, scoring 0.469, just below CodeBERT's 0.480, both significantly outperforming DeepMutation.

TABLE III
(RQ2) AVERAGE SEMANTIC SIMILARITY FOR EACH INTEGRATION.

	Major	Major + CodeBERT	Major	Major + DeepMutation	Major	Major + MIN
#faults	285	285	41	41	434	434
#mutants	138142	177757(+29%)	17981	18200(+1%)	205008	232537(+13%)
#mutants/fault	484.7	623.7(+29%)	438.5	443.9(+1%)	472.3	535.8(+13%)
Avg. Ochiai	0.713	0.802(+12%)	0.686	0.723(+5%)	0.721	0.798(+11%)

B. RQ2: Gains of integration into Major

Since RQ1 shows that CodeBERT slightly outperforms the others in defect simulation, we are further interested in understanding the complementary strengths of these tools when combined with an established baseline. Specifically, we ask: how much additional benefit does each tool provide when its mutants are added to those of Major? To answer this, we compare the performance of the following combinations: MIN + Major, CodeBERT + Major, and DeepMutation + Major.

Table III presents the semantic similarity data. It is important to note that during the merging process of the two kill matrices, we filtered out non-compilable mutants generated by CodeBERT, DeepMutation, and MIN. This step was crucial as non-compilable mutants should not contribute to the similarity calculation. As observed from the table, DeepMutation exhibits poor practicality, being applicable to only a few projects and generating a limited number of mutants, thereby providing a constrained boost to Ochiai similarity. On the other hand, CodeBERT significantly enhances similarity, albeit at the cost of introducing a substantial number of additional mutants. Conversely, combining MIN with Major emerges as the optimal choice, striking a balance between the number of mutants introduced and the degree of similarity enhancement.

Figure 7 shows boxplots that illustrate the diversity. Major + MIN + CodeBERT + DeepMutation is not included in the figure as it represents the highest possible diversity (OP=1). From the figure, it can be observed that any integration methods result in significant gains in diversity, with the addition of MIN offering a more pronounced boost. Diversity(Major + MIN) = 97.5% is greater than Diversity(Major + CodeBERT) = 96.7%, which is greater than Diversity(Major + DeepMutation) = 95.4%, and Diversity(Major) = 94.4%. This suggests that MIN-generated mutants can better simulate defects that cannot be simulated by existing operators.

To further support these observations, we conducted statistical hypothesis testing to evaluate the significant differences among the four approaches. A Wilcoxon signed-rank test was performed, and the p-values were adjusted using the Benjamini-Hochberg method to control the false discovery rate. Additionally, we calculated Cliff's Delta to measure the effect size. The results are summarized in Table IV.

From Table IV, we can observe the following:

- Major + MIN vs Others: The p-values for comparisons involving Major + MIN are all statistically significant (adjusted p-value less than 0.01), with Cliff's Delta values indicating medium to large effect sizes. This underscores the significant diversity improvement achieved by integrating MIN.

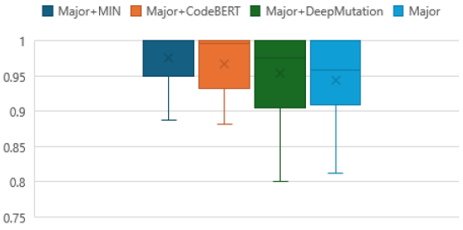

Fig. 7. (RQ2) Diversity for each integration.

- Major + CodeBERT vs Major + DeepMutation: The comparison between Major + CodeBERT and Major + DeepMutation has a smaller adjusted p-value (0.005176) and a relatively low effect size (Cliff's Delta = 0.108033), suggesting that the two methods provide somewhat similar contributions to diversity.
- Major + DeepMutation vs Major: The adjusted p-value for this comparison (0.698056) indicates no statistically significant difference, with a negligible effect size (Cliff's Delta = 0.066482). This suggests that DeepMutation does not contribute significantly to diversity when compared to Major.

If we combine the results of RQ1, it becomes evident that while CodeBERT performs well as a single operator, its flaws in simulations are often coupled by Major. Therefore, the additional benefit from integrating CodeBERT into Major is smaller compared to MIN.

Conclusion: RQ2 - Gains of Integration into Major

The integration of MIN, CodeBERT, and DeepMutation into Major reveals distinct trade-offs. CodeBERT achieves the highest semantic similarity (0.802) but increases mutant generation by 29%, raising efficiency concerns. DeepMutation offers a modest similarity gain (+5%) with fewer mutants, but its limited applicability makes it less practical. MIN, on the other hand, strikes a balance, improving semantic similarity to 0.798 with only a 13% increase in mutants. It also provides the highest diversity gain, outperforming both CodeBERT and DeepMutation, making it a more efficient and effective addition to Major.
Thus, MIN offers the best balance between defect simulation, mutant generation, and diversity, positioning it as the most practical choice for mutation testing in real-world scenarios.

TABLE IV
(RQ2) HYPOTHESIS TESTING RESULTS FOR DIVERSITY.

Comparison	p-value	Cliff's Delta	Adjusted p-value	Effect Size
Major + MIN vs Major + CodeBERT	**0.000433**	**0.227147**	**0.000876**	**small**
Major + MIN vs Major + DeepMutation	**0.000438**	**0.310249**	**0.000876**	**small**
Major + MIN vs Major	**0.000437**	**0.379501**	**0.000876**	**medium**
Major + CodeBERT vs Major + DeepMutation	**0.004313**	0.108033	**0.005176**	negligible
Major + CodeBERT vs Major	**0.000999**	**0.193906**	**0.001498**	**small**
Major + DeepMutation vs Major	0.698056	0.066482	0.698056	negligible

C. RQ3: Efficiency of defect simulation

TABLE V
(RQ3) GENERATION TIME FOR EACH MUTATOR.

	CodeBERT	MIN	DeepMutation
#of all mutants	342444	42619	114703
total generation time(s)	145025.618	11528.841	81732.039
average time to generate a mutant(s)	0.424	0.271	0.713
average time to generate a compilable mutant(s)	3.655	0.362	1.500

After analyzing RQ1 and RQ2, we arrive at several key conclusions: 1. CodeBERT, when used as a standalone operator, marginally outperforms MIN in defect simulation; 2. DeepMutation demonstrates the poorest performance among the evaluated options; and 3. The combination of MIN and Major yields slightly superior defect simulation capabilities compared to CodeBERT and Major.

Moving forward, we shift our focus to comparing the efficiency of different operators. If the defect simulation capabilities are comparable, we will place greater emphasis on the efficiency of the operators. Specifically, we examine the efficiency by analyzing the average time taken by each operator from method initiation to mutant file generation. It is important to note that we exclude the time taken for mutants to execute, as non-compilable mutants are significantly with less execution time. However, the practical utility of non-compilable mutants is negligible. Additionally, we provide supplementary information by reporting the percentage of non-compilable mutants for each operator.

Table V compares the time required to generate mutants by MIN, CodeBERT, and DeepMutation. Notably, MIN demonstrates superior efficiency, with the average time for generating a single mutant being 64% of CodeBERT's time and only 38% of DeepMutation's time. Despite parsing the entire project, MIN's focus on generating mutants for a single class in our settings. It is worth mentioning that the average time spent by MIN would be even lower if mutants were generated for the entire project. DeepMutation, although needing to parse the entire project, already includes mutants for the entire project in the data presented. On the other hand, CodeBERT's time for generating individual mutants remains relatively consistent across projects, owing to its fixed-length code snippet approach. For Major, the option `-J-Dmajor.export.mutants=true` is required to generate mutant files using the Java compiler embedded within Major. However, this approach may lead to failures in compiling the target programs since the different Java version. As a result, we did not include Major in the formal comparisons in RQ3. Nevertheless, our experimental data indicates that the average time taken by Major to successfully generate a single mutant across different projects ranges from 0.006s to 0.138s.

In Table VI, we compare the ratio of uncompilable mutants as supplementary information. The majority of mutants generated by CodeBERT are invalid, while nearly half of DeepMutation's mutants fail to compile. In contrast, MIN exhibits relatively high-quality mutants, comparable to traditional operators in terms of the percentage of mutants that pass compilation. Our further analysis indicates that these non-compilable mutants primarily stem from challenges in type inference and resolving inheritance relationships. For example, in the case of **Math-70** (See more details in Section VII.D), the method with argument inherited from the parent class, making it difficult for MIN to determine the type of the actual parameter when type inference fails. As a result, it generates an invalid mutant, leading to compilation failure. While more precise type inference could mitigate such errors, it comes with significant computational overhead. Therefore, MIN makes a trade-off between accuracy and efficiency in its design.

Overall, MIN emerges as a promising approach, showcasing efficient mutant generation with high-quality mutants, positioning it as a competitive option for mutation testing in software quality assurance.

Conclusion: RQ3 - Efficiency of Defect Simulation

RQ3 analysis shows that MIN excels in mutant generation efficiency, requiring only 64% of the time taken by CodeBERT and 38% of DeepMutation. Additionally, MIN generates high-quality mutants, with compilable mutants produced in just 10% of the time taken by CodeBERT and 24% of DeepMutation on average. These results make MIN a promising choice for mutation testing, offering faster mutant generation without compromising quality. Integrating MIN into existing frameworks can significantly improve both efficiency and defect simulation.

D. RQ4: Fault detection capability with subsumption

Table VII presents the number of faults detected by each mutator using the minimal test suite approach. Due to the stochastic nature of generating the minimal set, the detected defects may not be integer values. For instance, if the minimal set is generated 20 times for a single defect and two of these

TABLE VI
(RQ3) UNCOMPILABLE RATIO FOR EACH MUTATOR.

	COR	ROR	LVR	STD	AOR	ORU	LOR	SOR	CodeBERT	MIN	DeepMutation
#of all mutants	34210	53904	54363	30743	55748	1416	2564	1736	342444	42619	462
#of uncompilable mutants	8125	13850	16734	7049	15433	186	766	407	302829	10590	243
uncompilable ratio	0.238	0.257	0.308	0.229	0.277	0.131	0.303	0.234	0.884	0.252	0.525

TABLE VII
(RQ4) DETECTED FAULTS BY MINIMAL TEST SUITE.

	DeepMutation	CodeBERT	MIN
# of analyzed faults	44	285	434
# of detected faults	5.2	111.75	187
Detected ratio	11.8%	39.2%	42.9%

TABLE VIII
(RQ4) UNIQUELY DETECTED FAULTS BY SUBSUMING MUTANTS.

	MIN	CodeBERT
# of undetected faults by Major	52	52
# of new detected faults	20	16
Detected ratio	38.5%	30.8%

instances detect the defect, then the minimal set detects 0.1 defects for that particular case. In other words, using the minimal set approach, the defect is detected with a probability of 0.1. As can be seen from the table, defects are more likely to be detected using MIN.

Next, we examine the intersection of two criteria: 1. Cases where Major fails to detect defects (i.e., triggering test cases do not yield unique mutation score boosts); and 2. Instances where a new mutator can generate at least one subsuming mutant that is exclusively killed by triggering test cases.

Initially, we identified the 19 bugs common to MIN, CodeBERT, and DeepMutation (as illustrated in Figure 4). However, it was observed that DeepMutation failed to generate mutants satisfying criterion "2" for any of these bugs, indicating its ineffectiveness. Consequently, DeepMutation was excluded from further comparison. Subsequently, we expanded the bug set to encompass the 289 projects compatible with CodeBERT. Table VIII presents the findings. Among these 289 bugs, 52 satisfied criterion "1". Among these bugs, MIN and CodeBERT generated at least one mutant satisfying criterion "2" for 20 and 16 bugs, respectively. Thus, MIN exhibits a superior ability to generate subsuming mutants capable of detecting defects: integrating MIN into Major can reduce the number of undetected defects by 38.5%.

Conclusion: RQ4: Fault Detection capability with subsumption

In examining the fault detection capability with subsumption, MIN demonstrates significant promise. Leveraging the minimal test suite approach, MIN detects defects with a ratio of 42.9%, outperforming both CodeBERT and DeepMutation. Furthermore, when considering uniquely detected faults by subsuming mutants, MIN's integration into Major offers substantial advantages, reducing the number of undetected faults by 38.5%. This underscores MIN's efficacy in enhancing fault detection capabilities within the mutation testing framework.

VI. RELATED WORK

Defect injection encompasses two primary approaches. The first involves mutators in mutation testing, which inject defects according to specific rules. The second relies on learning-based methods, where defects are injected based on trained models. In traditional mutators, the operators targeting method calls are relatively limited. Offutt discovered that using just five specific mutators achieves the same effectiveness as using all mutators [26]: ABS, AOR, LCR, ROR, and UOI. Notably, none of these mutators directly focus on method invocation, despite a considerable proportion of real defects being related to method invocation. This highlights the urgent need for improving the ability of mutators to model defects related to method calls in mutation testing.

Rule-based mutation. IBIR [23] is a framework designed to automatically inject defects based on bug reports. The process involves using the bug report to identify a suitable injection location, followed by the injection itself using manually predefined patterns. Khanfir et al. also discussed the method invocation pattern, but gave no detailed explanations on how to match feasible method names. Furthermore, their implementation [27] does not provide a specific implementation for the "MethodInvocationMutator," and none of the 181 mutants generated for the three example projects were produced by this mutator. As a result, IBIR shows limitations in generating defects related to method calls. Khanfir et al. reported a semantic similarity below 0.6 for all types of defects, suggesting room for improvement. To enhance the results of IBIR, we propose integrating a concrete implementation of MIN. By doing so, the effectiveness of defect injection related to method calls could be significantly improved. For our study, we opted not to include IBIR as a baseline due to its reliance on additional tools such as bug reports and defect location tools. This dependence introduces potential complexities and variations in the results, which could affect the comparative analysis.

Learning-based mutation. In the field of learning-based mutators, DeepMutation and CodeBERT have emerged as two prominent approaches. Their fundamental concept involves converting code into vectors using pre-trained word embeddings, which are then used to train and make predictions using

a learning model. However, a significant challenge arises from the fact that these pre-trained word embeddings might not be well-suited to the specific project being analyzed. Consequently, they may struggle to perform effectively with highly unique tokens present in the project's codebase. SeemSed tries to address this issue to some extent by crawling the top 1000 frequently occurring words from external projects. However, this approach often fails to include unique method names, leading to their exclusion from the prediction results. Moreover, integrating complex learning models seamlessly into existing mutation testing tools proves challenging as a single mutator. This integration process necessitates vectorizing the source program first and then predicting mutants individually. Unfortunately, this approach introduces significant additional overhead and raises concerns about potential overlap with traditional mutators.

VII. DISCUSSION

A. The size effect

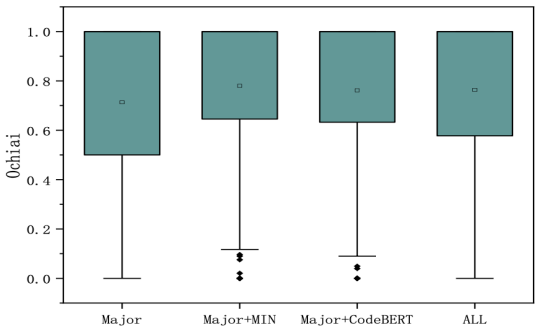

Fig. 8. Semantic similarity for Major + MIN and Major + CodeBERT when size is controlled.

In RQ1 and RQ2, we observed that when the size is not controlled, the performance on semantic similarity of MIN and CodeBERT is relatively similar. In this section, we delve deeper into whether controlling the size of generated mutants alters this conclusion.

In essence, we investigate the size effect: assuming computational resources are limited to 100 newly generated mutants, which approach—MIN or CodeBERT—is more effective to integrate?

As a result, the practical scenario imposes size control on the total number of generated mutants, regardless of their compilability. This introduces the possibility of uncompilable mutants being included within the 100 sampled mutants. Figure 8 demonstrates that Major + MIN achieves the highest semantic similarity, outperforming alternatives.

To conclude, when the size of generated mutants is controlled, MIN demonstrates more pronounced advantages, especially when the cost of filtering uncompilable mutants is taken into account. This highlights the practical efficiency and effectiveness of MIN, making it a superior choice for resource-constrained scenarios.

B. Implications for developers

The workflow of using MIN is consistent with traditional mutation testing: developers first conduct mutation testing by executing tests and identifying surviving mutants. For these surviving mutants, additional test cases are specifically designed to attempt to kill them. The value of MIN in this process lies in evaluating whether the new test cases designed for MIN mutants are *triggering*, meaning they can detect real faults.

Even though developers may not know the real bugs in their code upfront, the effectiveness of MIN can be assessed through our experimental setup, which compares against known real-world bugs. If MIN performs well on benchmarks like Defects4J, it is reasonable to believe that MIN will also generate meaningful mutants in real-world settings.

Meanwhile, MIN is assessed based on its ability to generate *subsuming mutants*—mutants that reveal faults in ways other operators cannot. This ensures that MIN adds unique value beyond traditional mutation operators.

This aspect is reflected in our experiments, particularly in **RQ4 (Fault Detection Capability with Subsumption)**. Table VIII highlights the ability of mutation operators to generate subsuming mutants tied to real faults:

1) We identified projects where Major operators fail to detect faults (i.e., triggering test cases do not yield unique mutation score boosts). This implies that the mutants generated by Major are not fully coupled to the real faults.
2) For these projects, we examined whether MIN could generate subsuming mutants that are killed by triggering tests (indicating coupling to real faults) and not by any other tests (ensuring subsumption).

Our results show that in these challenging scenarios, MIN produced at least one subsuming mutant for 38.5% of these projects, significantly outperforming CodeBERT, which achieved only 30%.

To summarize, even when developers are unaware of specific bugs in their code, MIN can be used in the same way as traditional mutation testing. If we assume that the distribution of faults in real-world scenarios is similar to Defects4J, MIN demonstrates a clear advantage:

- Using **Major alone**: 18.0% (52/289) of defects remain undetected.
- Using **Major + MIN**: Only 11.1% (32/289) of defects remain undetected.
- Using **Major + CodeBERT**: 12.5% (36/289) of defects remain undetected.

MIN outperforms CodeBERT while generating fewer mutants, making it more efficient and practical for developers. This suggests that integrating MIN into mutation testing pipelines can significantly enhance fault detection capability and efficiency in real-world software development scenarios.

C. Should uncompilable mutants be labeled as killed?

As is well-known, Major marks uncompilable mutants as "killed", which is a common practice in mutation testing. The

reason for this is that uncompilable mutants, which do not contribute meaningfully to testing, are typically not considered useful by developers. The focus is on surviving mutants, which are the ones that could potentially indicate bugs or weaknesses in the system. However, in this study, we treat uncompilable mutants as "survived" for MIN, CodeBERT, and DeepMutation.

This decision stems from the observation that only "killed" mutants can contribute to the evaluation metrics, specifically semantic similarity and diversity. Surviving mutants, by definition, do not offer any improvement to these metrics. For example, if an uncompilable mutant is marked as "killed", it could artificially inflate the semantic similarity score, which would not accurately reflect the practical utility of that mutant. Therefore, marking uncompilable mutants as "survived" ensures that they do not contribute any false positives to the metrics. In CodeBERT's paper [8], they filtered out uncompilable in advance so that uncompilable mutants do not participate in the calculation of evaluation metrics. In evaluation, it is actually equivalent to marking them as alive.

From our point of view, the contradiction arises from two conflicting statements:

1. Developers should pay attention to surviving mutants, so uncompilable mutants, which are essentially useless, should be marked as "killed".

2. In mutation testing evaluation, only "killed" mutants can actually increase the evaluation indicators (e.g., semantic similarity and diversity). Therefore, uncompilable mutants, which cannot contribute meaningfully, should not be labeled as "killed."

However, the consensus remains that uncompilable mutants have minimal practical value, and marking them as "killed" is an accepted practice. While this approach is widely followed, we recognize the potential overestimation of Major's performance, as uncompilable mutants marked as "killed" can artificially boost metrics.

Why We Did Not Alter Major's Setting?

We did not change Major's default handling of uncompilable mutants because this setting is almost universally accepted in mutation testing research. The practice of marking uncompilable mutants as "killed" is entrenched in most studies and tools, including PIT, which is a widely used mutation testing framework. By retaining Major's default setting, we align with the established norms in the community, allowing for a fair comparison with previous work.

However, it is important to note that this choice may lead to an overestimation of Major's effectiveness in RQ1 and RQ2, as the marking of uncompilable mutants as "killed" could inflate its performance. We hope that readers understand this inherent bias when interpreting the results for Major. However, even so, MIN demonstrates strong competitiveness, suggesting its advantages would be even greater with labeling uncompilable as alive for Major.

In conclusion, while we acknowledge the trade-offs involved in labeling uncompilable mutants as "survived" for MIN, CodeBERT, and DeepMutation, we believe that this approach better reflects the practical value of mutants in mutation testing. Although Major's performance may be slightly overestimated due to its default settings, we maintain this approach to ensure consistency with prior research and to avoid undermining Major's results unfairly.

D. Case Study

In [22], the authors analyzed the types of defects that Major is unable to simulate (i.e., triggering test cases kill no mutant) and provided 13 case studies. Among these, Math-34 and Closure-11 were excluded due to failures or out-of-time in running MIN. Meanwhile, we have not found Lang-658 in Defects 2.0.0. As a result, we present an in-depth analysis and discussion of the remaining 10 cases.

1) Defects successfully simulated or subsumed by MIN:

- **Chart-16:** In this case, the code snippet above shows the difference between the buggy version (indicated with '-') and the fixed version (indicated with ' + ')

```
@@ -204,8 +204,8 @@
                }
                else {
+                   this.seriesKeys = new
    Comparable[0];
+                   this.categoryKeys = new
    Comparable[0];
-                   this.seriesKeys = null;
-                   this.categoryKeys = null;
                }
            }
@@ -335,7 +335,7 @@
            if (categoryKeys == null) {
                throw new
                    IllegalArgumentException("
                    Null 'categoryKeys'
                    argument.");
            }
+           if (categoryKeys.length !=
    getCategoryCount()) {
-           if (categoryKeys.length != this.
    startData[0].length) {
                throw new
                    IllegalArgumentException(
                        "The number of
                            categories does
                            not match the data
                            .");
            }
```

In this case, the 56th mutant generated by MIN subsumes the original defect. The only test case that killed this mutant was one of the eight triggering test cases. Based on the current test suite, this indicates that if we kill this mutant, we must include one of the triggering test case.

- **Math-75:**

```
     */
    @Deprecated
    public double getPct(Object v) {
+       return getPct((Comparable<?>) v);
-       return getCumPct((Comparable<?>)
    v);
    }

    /**
```

The 30th mutant generated by MIN is identical to the original defect.

- **Closure-92:**

```
@@ -786,7 +786,7 @@
       } else {
         // In this case, the name was
             implicitly provided by two
             independent
         // modules. We need to move this
             code up to a common module.
+         int indexOfDot = namespace.
    lastIndexOf('.');
-         int indexOfDot = namespace.
    indexOf('.');
         if (indexOfDot == -1) {
           // Any old place is fine.
           compiler.
               getNodeForCodeInsertion(
               minimumModule)
```

The 584th mutant generated by MIN is identical to the original defect.

2) Defects partially coupled with the MIN mutants:

- **Lang-4:**

```
@@ -28,7 +28,7 @@
  */
 public class LookupTranslator extends
     CharSequenceTranslator {

+    private final HashMap<String,
    CharSequence> lookupMap;
-    private final HashMap<CharSequence,
    CharSequence> lookupMap;
     private final int shortest;
     private final int longest;
@@ -43,21 +43,21 @@
      * @param lookup CharSequence[][]
          table of size [*][2]
      */
     public LookupTranslator(final
         CharSequence[]... lookup) {
+        lookupMap = new HashMap<String,
    CharSequence>();
-        lookupMap = new HashMap<
    CharSequence, CharSequence>();
         int _shortest = Integer.MAX_VALUE
             ;
         int _longest = 0;
         if (lookup != null) {
             for (final CharSequence[] seq
                  : lookup) {
                 this.lookupMap.put(seq
                     [0].toString(), seq
                     [1]);
                 final int sz = seq[0].
                     length();
                 if (sz < _shortest) {
                     _shortest = sz;
                 }
                 if (sz > _longest) {
                     _longest = sz;
                 }
             }
         }
         shortest = _shortest;
         longest = _longest;
@@ -74,7 +74,7 @@
         // descend so as to get a greedy
             algorithm
         for (int i = max; i >= shortest;
             i--) {
             final CharSequence subSeq =
                 input.subSequence(index,
                 index + i);
+            final CharSequence result =
    lookupMap.get(subSeq.toString());
-            final CharSequence result =
    lookupMap.get(subSeq);
             if (result != null) {
                 out.write(result.toString
                     ());
                 return i;
```

In this case, the second mutant generated by MIN, where `this.lookupMap.put(seq[0].toString(), seq[1])` was modified to `replace` instead of `put`, partially coupled with the defect. This change impacted the behavior of the `lookupMap` in handling `LookupTranslator`, which in turn influenced the defect's manifestation. Among the 14 test cases that failed, 1 was able to reveal the defect.

- **Math-11:**

```
@@ -180,7 +180,7 @@
             throw new
                 DimensionMismatchException
                 (vals.length, dim);
         }

+        return FastMath.pow(2 * FastMath.
    PI, -0.5 * dim) *
-        return FastMath.pow(2 * FastMath.
    PI, -dim / 2) *
             FastMath.pow(
                 covarianceMatrixDeterminant
                 , -0.5) *
             getExponentTerm(vals);
```

The 19th mutant replaced `FastMath.pow(2 * FastMath.PI, -0.5 * dim)` with `FastMath.atan2(2 * FastMath.PI, -0.5 * dim)` and obtained three failed test cases, one of which revealed the defect.

- **Math-94:**

```
@@ -409,7 +409,7 @@
      * @since 1.1
      */
     public static int gcd(int u, int v) {
+        if ((u == 0) || (v == 0)) {
-        if (u * v == 0) {
             return (Math.abs(u) + Math.
                 abs(v));
         }
```

The 64th mutant replaced `return (Math.abs(u) + Math.abs(v));` with `return (Math.abs(u) + Math.negateExact(v));` and obtained three failed test cases, one of which revealed the defect.

This case is particularly interesting because the mutant was not directly injected at the location where the error occurs. However, it still produced a highly relevant mutant: an experienced programmer would likely consider

designing test cases involving positive values or overflow scenarios to distinguish the mutant from the original program. Furthermore, they might also notice the potential risk of overflow when calculating `u * v`, leading to a deeper understanding of the defect and its triggering conditions.

- **Time-4:**

```
@@ -461,7 +461,7 @@
             System.arraycopy(iValues, i,
                 newValues, i + 1,
                 newValues.length - i - 1);
             // use public constructor to
                 ensure full validation
             // this isn't overly
                 efficient, but is safe
+            Partial newPartial = new
    Partial(newTypes, newValues,
    iChronology);
-            Partial newPartial = new
    Partial(iChronology, newTypes,
    newValues);
             iChronology.validate(
                 newPartial, newValues);
             return newPartial;
         }
```

The first mutant by MIN performs a method call replacement within the constructor of `Partial`, allowing it to partially couple with the real defect. Among the 23 test cases that fail due to this mutant, one is a triggering test case for the actual defect.

However, the current MIN does not yet target constructors, as constructors and other method calls can be considered two separate areas. In the future, we plan to extend MIN to handle constructors as well.

3) Defects that MIN failed to simulate or resulted in compilation failures:

- **Chart-10:**

```
@@ -62,7 +62,7 @@

      * @return The formatted HTML area
          tag attribute(s).
      */
     public String generateToolTipFragment
         (String toolTipText) {
+        return " title=\"" +
    ImageMapUtilities.htmlEscape(
    toolTipText)
-        return " title=\"" + toolTipText
             + "\" alt=\"\"";
     }
```

Here, `ImageMapUtilities.htmlEscape` is a static method. The corresponding class, `ImageMapUtilities`, contains only one static method with a string parameter and a string return type, leaving no candidates for replacement.

- **Lang-51:**

```
@@ -679,7 +679,6 @@
                 (str.charAt(1) == 'E' ||
                     str.charAt(1) == 'e')
                     &&
                     (str.charAt(2) == 'S' ||
                         str.charAt(2) == 's');
         }
-        return false;
         }
         case 4: {
             char ch = str.charAt(0);
```

Although the context involves the method call `str.charAt(0)`, the `String` class in the library has only one method with an integer parameter and a character return type, leaving no candidates for replacement.

- **Math-70:**

```
@@ -69,7 +69,7 @@

         throws
             MaxIterationsExceededException
             , FunctionEvaluationException
             {
+        return solve(f, min, max);
-        return solve(min, max);
```

The non-compilable mutant generated by MIN was `solve(max, min, f)`. This mutant was created due to a misjudgment of `f`'s type. The type could not be determined because `f` is a member variable inherited from a parent class, and multiple overloaded `solve` methods are defined in the class. For efficiency reasons, MIN does not trace back to parent classes to infer the type. Without accurate type information in polymorphic contexts, MIN defaults to using the first matching method signature based on parameter count, leading to the compilation error.

4) Summary: The case study analysis demonstrates the effectiveness and limitations of MIN across 10 representative defects identified in [23]. In 7 of these cases, MIN successfully enhanced Major's effectiveness by either fully replicating, subsuming, or partially coupling with the real defects. Specifically: Chart-16, Math-75, Closure-92 Lang-4, Math-11 Math-94, and Time-4. Overall, the findings underline MIN's significant contribution to advancing mutation testing by expanding the scope and precision of defect simulation. By successfully addressing 7 out of the 10 cases, MIN demonstrates its capability to model complex defects that remain challenging for existing tools like Major, validating its role as a meaningful enhancement in the mutation testing landscape.

VIII. THREATS

The primary threat to our study revolves around the confounding effect of mutant size. A key question arises: does an operator's high effectiveness stem from its inherent ability to simulate defects, or is it simply a result of the large number of mutants it generates? While we acknowledge this potential concern, directly controlling the number of generated mutants presents challenges. The volume of generated mutants is inherently tied to a mutation operator's capacity. For example, operators like DeepMutation or Major's SOR exhibit limitations in defect generation, applying only to a small number of cases. Imposing an artificial constraint—such

as capping all operators at 10 mutants per defect—would unfairly restrict more capable operators while leaving less effective ones unaltered.

To mitigate this threat, we designed our experiments to separate these dimensions as much as possible. When evaluating defect simulation and detection capabilities, we ensured fairness by assessing every mutant generated by each operator, regardless of quantity. While generating more mutants can be advantageous, it also reflects an operator's strength in defect simulation. However, this is not always the case; as shown in Table III, AOR produces significantly more mutants than MIN yet exhibits lower similarity to real defects. To account for scale effects, we introduced practical efficiency as an evaluation metric. By reporting the elapsed time required for mutant generation, readers can better assess the efficiency of each mutation operator. This approach allows us to navigate the complexities of mutant size while maintaining a fair and comprehensive evaluation.

Additionally, we further discuss the impact of mutant size in Section VII.A, particularly in cases where CodeBERT and MIN perform similarly. Our results indicate that when size is controlled, MIN demonstrates a clear advantage, reinforcing its effectiveness beyond mere volume.

Another threat stems from our experimental scope: we generate mutants only for the classes (that were) modified by the bug fix, rather than the entire project. While this approach is consistent with prior work and motivated by practical considerations—e.g., applying CodeBERT to the entire project would produce an overwhelming number of mutants—it may introduce a potential underestimation of the tools' full capabilities. In some cases, generating mutants for the entire codebase could lead to stronger fault detection or higher diversity. This limitation, while partially offset by realism and comparability, should be acknowledged in interpreting the generalizability of our results.

Beyond the size and scope effects, our approach has another limitation: it requires access to the source code of software and libraries to construct the method HashMap, a crucial part of the parsing process. While this requirement does not affect the validity of our results or the general applicability of our technique, it is an important consideration for tool builders and researchers implementing similar approaches.

Another limitation lies in the scope of defects considered in our study. Our evaluation is based on a relatively small set of 19 defects, which may not fully capture the diversity of real-world software faults. As a result, while our findings highlight clear trends, they should be interpreted with this limitation in mind. Future studies incorporating a larger and more diverse set of defects would further validate and strengthen our conclusions.

Finally, MIN currently does not generate mutants for class constructors, which may restrict its applicability in scenarios where constructor-level mutations are significant. Additionally, MIN faces challenges when handling generic methods, potentially limiting its ability to generate mutants for all method types. These constraints should be considered when applying MIN to projects that heavily rely on constructors or generics, as they may impact the technique's generalizability.

IX. CONCLUSION

In conclusion, this study addresses the critical need for improved defect simulation in mutation testing by introducing the innovative MIN (Method INvocation mutator). The experimental results underscore MIN's effectiveness in enhancing defect simulation capabilities for real defects. Integration of MIN into Major significantly improves semantic similarity to real defects by 11%, enhances the diversity of mutant sets up to 97.5%, and reduces undetected faults by 38.5%. Moreover, MIN exhibits remarkable efficiency and fault detection capability, being 10 times faster than CodeBERT and 4 times faster than DeepMutation in generating individual compilable mutants.

Furthermore, in the context of the burgeoning interest in advanced machine learning-based mutation operators like DeepMutation and CodeBERT, this study demonstrates that integrating MIN into Major enables it to achieve defect simulation capabilities comparable to integrating the state-of-the-art CodeBERT mutator into Major. MIN + Major exhibits performance metrics in terms of semantic similarity and diversity that are on par with, if not surpassing, those achieved by CodeBERT + Major.

These findings highlight the significance of MIN as a valuable addition to the mutation testing toolkit. By offering efficient mutant generation, MIN has the potential to revolutionize defect simulation in software quality assurance. Integrating MIN into existing mutation generation tools would not only enhance defect simulation capabilities but also streamline the mutation testing process, ultimately leading to improved software quality and reliability.

REFERENCES

[1] DARPA CGC. 2018. Darpa Cyber Grand Challenge (CGC) Binaries. [Online]. Available: https://github.com/CyberGrandChallenge/
[2] B. D. Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. K. Robertson, F. Ulrich, and R. Whelan, "LAVA: Large Scale Automated Vulnerability Addition," in *S&P 2016*, pp. 110–121, 2016.
[3] A. Habib and M. Pradel, "How Many of All Bugs Do We Find? A Study of Static Bug Detectors," in *ASE 2018*, pp. 317-328.
[4] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649-678, 2011.
[5] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, and B. Xu, "How far we have progressed in the journey? An examination of cross-project defect prediction," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 1, pp. 1-51, 2018.
[6] E. Winter, V. Nowack, D. Bowes, S. Counsell, T. Hall, S. Ó. Haraldsson, and J. R. Woodward, "Let's Talk With Developers, Not About Developers: A Review of Automatic Program Repair Research," *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 419-436, 2023.
[7] J. Patra and M. Pradel, "Semantic bug seeding: a learning-based approach for creating realistic bugs," in *FSE 2021*, pp. 906-918.
[8] R. Degiovanni and M. Papadakis, "μBert: Mutation Testing using Pre-Trained Language Models," in *ICST Workshops 2022*, pp. 160-169.
[9] M. Tufano, J. Kimko, S. Wang, C. Watson, G. Bavota, M. D. Penta, and D. Poshyvanyk, "DeepMutation: A Neural Mutation Tool," in *ICSE (Companion Volume) 2020*, pp. 29–32.
[10] H. Osman, M. Lungu, and O. Nierstrasz, "Mining frequent bug-fix code changes," in *CSMR-WCRE 2014*, pp. 343-347.
[11] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: a database of existing faults to enable controlled testing studies for Java programs," in *ISSTA 2014*, pp. 437-440.
[12] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *ASE 2011*, pp. 612-615.

[13] Defects4J. [Online]. Available: https://github.com/rjust/defects4j
[14] Y. Ma, J. Offutt, and Y. R. Kwon, "MuJava: a mutation system for java," in *ICSE 2006*, pp. 827-830.
[15] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for Java (demo)," in *ISSTA 2016*, pp. 449-452.
[16] PIT. [Online]. Available: http://pitest.org/
[17] PIT mutators. [Online]. Available: http://pitest.org/quickstart/mutators
[18] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *NAACL-HLT (1) 2019*, pp. 4171-4186.
[19] R. Hosseini and P. Brusilovsky, "JavaParser; A Fine-Grain Concept Indexing Tool for Java Problems," in *AIED Workshops 2013*.
[20] JavaParser. [Online]. Available: https://javaparser.org/
[21] M. Ojdanic, A. Garg, A. Khanfir, R. Degiovanni, M. Papadakis and Y. L. Traon, "Syntactic Versus Semantic similarity of Artificial and Real Faults in Mutation Testing Studies," *IEEE Transactions on Software Engineering*, 2023. DOI: 10.1109/TSE.2023.3277564
[22] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *FSE 2014*, pp. 654-665.
[23] A. Khanfir, A. Koyuncu, M. Papadakis, M. Cordy, T. F. Bissyande, J. Klein, and Y. L. Traon, "IBIR: Bug Report driven Fault Injection," *ACM Transactions on Software Engineering and Methodology*, 2022.
[24] P. Zhang, Y. Wang, X. Liu, Y. Li, Y. Yang, Z. Wang, X. Zhou, L. Chen, and Y. Zhou, "Mutant reduction evaluation: what is there and what is missing?" *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 4, pp. 69:1-69:46, 2022.
[25] 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 *EMNLP (Findings) 2020*, pp. 1536-1547.
[26] J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99-118, 1996.
[27] IBIR. [Online]. Available: https://github.com/serval-uni-lu/IBIR
[28] Z. Tian, J. Chen, Q. Zhu, J. Yang, and L. Zhang, "Learning to Construct Better Mutation Faults," in *ASE 2022*, pp. 64:1-64:13.
[29] M. E. Delamaro, J. Offutt, and P. Ammann, "Designing deletion mutation operators," in *ICST 2014*, pp. 11-20.
[30] M. E. Delamaro, L. Deng, V. H. S. Durelli, N. Li, and J. Offutt, "Experimental evaluation of SDL and one-op mutation for C," in *ICST 2014*, pp. 203-212.
[31] G. Guizzo, F. Sarro, J. Krinke, and S. Vergilio, "Sentinel: A hyper-heuristic for the generation of mutant reduction strategies," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 803-818, 2022.
[32] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *ICST 2010*, pp. 90-99.
[33] "MIN: Method INvocation mutator," 2025. [Online]. Available: https://github.com/zhangpengNJU/MIN2025
[34] C. O. Fritz, P. E. Morris, and J. J. Richler, "Effect size estimates: current use, calculations, and interpretation," *Journal of experimental psychology: General*, vol. 141, no. 1, p. 2, 2012.