

# KATANA: Dual Slicing-Based Context for Learning Bug Fixes

MIFTA SINTAHA, The University of British Columbia, Canada

NOOR NASHID, The University of British Columbia, Canada

ALI MESBAH, The University of British Columbia, Canada

Contextual information plays a vital role for software developers when understanding and fixing a bug. Consequently, deep learning-based program repair techniques leverage context for bug fixes. However, existing techniques treat context in an arbitrary manner, by extracting code in close proximity of the buggy statement within the enclosing file, class, or method, without any analysis to find actual relations with the bug. To reduce noise, they use a predefined maximum limit on the number of tokens to be used as context. We present a program slicing-based approach, in which instead of arbitrarily including code as context, we analyze statements that have a control or data dependency on the buggy statement. We propose a novel concept called *dual slicing*, which leverages the context of both buggy and fixed versions of the code to capture relevant repair ingredients. We present our technique and tool called KATANA, the first to apply slicing-based context for a program repair task. The results show KATANA effectively preserves sufficient information for a model to choose contextual information while reducing noise. We compare against four recent state-of-the-art context-aware program repair techniques. Our results show KATANA fixes between 1.5 to 3.7 times more bugs than existing techniques.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**.

Additional Key Words and Phrases: program slicing, program repair, deep learning, contextual information, graph neural networks

## 1 INTRODUCTION

Traditional automated error detection [51] and program repair [8, 29, 32] techniques rely on a set of predefined templates and rules that are limited to specific software bug patterns; adding support for a new type of bug is manual and requires domain-specific knowledge in a given programming language. Instead of hard-coding error detection and repair patterns, we can automatically learn them from code examples of developer made mistakes and repairs, through deep learning [11, 16, 34, 41, 43].

To apply learning-based techniques for software analysis, the source code needs to be vectorized. It is, however, imperative to first delineate what information in the source code should be included as input for vectorization. While syntactical representation of source code for vectorization has gained traction in the literature in recent years, semantic information has received less attention. In particular, the notion of *context* pertaining to an erroneous statement in the code has been largely treated in an ad-hoc and arbitrary fashion. For instance, some techniques merely focus on the buggy statement [20, 49] and ignore context. Others use the enclosing file [12, 19], enclosing class [11], enclosing function [41, 57], or encapsulating AST subtrees [34] as context, often with a hard-coded bounding limit on the number of tokens.

Arbitrarily including code as context neither captures the true semantics of the buggy statement nor encompasses essential fix-ingredients. For developers, context plays a significant role in understanding a bug and determining a potential fix. This is true in case of machine learning models, where too much information in the input introduces noise that can affect the repair accuracy of

---

Authors' addresses: Mifta Sintaha, The University of British Columbia, Vancouver, Canada, msintaha@ece.ubc.ca; Noor Nashid, The University of British Columbia, Vancouver, Canada, nashid@ece.ubc.ca; Ali Mesbah, The University of British Columbia, Vancouver, Canada, amesbah@ece.ubc.ca.

2022. 1049-331X/2022/12-ART \$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

the model [17] and too little information can lead to overfitting or relevant data loss, leading to incorrect generation of patches. This poses questions pertaining to what is relevant context, how to collect it, what role the repair ingredients should play, and how relevant information related to both the erroneous and fixed code should be represented for deep learning consumption.

Finding adequate context ingredients is vital to overcoming this limitation. In fact, when a developer tries to fix a buggy line of code, they start from the buggy line and examine all the ingredients, such as the variables and function calls used in that line. They then go backward in the code, investigating where such ingredients have been defined, used, and modified throughout the code to understand the error. Our insight is that such relevant information can help a deep learning approach better reason about bug fixes. In particular, we propose to adopt backward slicing analysis for extracting contextual information in the form of code directly related to a buggy statement as well as its repaired ingredients, which we call *dual slicing* in this paper. Our approach, implemented in a tool called KATANA, extracts dual slicing-based context from the buggy and fixed files through inter and intraprocedural control and data flow analysis, transforms the slices to AST-based graph representations, and uses a Graph Neural Network (GNN) to train a model. To evaluate our dual slicing approach, we compare against four state-of-the-art repair techniques, and our results show that KATANA outperforms all four.

Our work makes the following contributions:

- A technique for extracting contextual information to enhance deep learning program repair. Our program slicing uses control and data flow analysis to find code that is relevant and applies the notion of *dual slicing* to capture context from both the buggy and fixed code. It requires no hard-coded bound on the number of tokens to limit the scope of the context.
- An evaluation of our dual slicing technique and comparison with four state-of-the-art learning-based program repair techniques. We train models on 91,181 pairs of buggy and fixed JavaScript files. KATANA achieves 42% ( $\frac{4,781}{11,397}$ ) repair accuracy within top-3 predictions, which is between 1.5 to 3.7 times higher than existing techniques.
- A quantitative analysis of the type and amount of information available for a learning model as well as a qualitative report on the type of bugs that KATANA can fix; 891 bugs are exclusively fixed by KATANA.
- The implementation of our technique, KATANA, which is available [4]; it includes our dataset, model and JavaScript program slicer, compatible with the latest JavaScript ES10.

## 2 MOTIVATION

Context is essentially the background information related to a particular problem depending on the domain. The importance of context has been discussed in natural language processing tasks [45] for establishing semantic similarity between words. For a human developer, context is key in understanding and creating a fix for a buggy program. Therefore, different types of context in both machine learning and rule-based approaches have been applied for fixing a bug [11, 34, 41, 61]. In case of a program repair, the context is typically considered as the surrounding source code relevant to the buggy statement. Current state-of-the-art learning based program repair approaches employ context in various ways and many of them use sequence-based and tree/graph based source code representation for learning program repair. These approaches use the buggy line with enclosing class [11], enclosing function [41, 57], buggy subtree [34], or the whole file [12] for capturing the context of a buggy line. However, all these approaches have a bounding limit for processing the buggy line. The sequence-based approaches use a maximum token limit (e.g., 1,000 tokens [11]), within which they extract buggy methods or classes to avoid truncated sequences. Similarly, current graph-based learning techniques use a maximum node limit (e.g., 500 nodes [12]) for program

repair. In practice, we cannot expect real world bugs to always be enclosed within small/medium sized methods or be limited to an arbitrary number of tokens or AST nodes. Therefore, this kind of quantitative constraint can potentially cause some important information pertaining to the bug fixes to be truncated and the overall context may even lose semantic meaning. In addition, increasing the limit on the number of tokens or nodes may add noise to the model.

```

1 'use strict';
2 import ENV from 'pass-ember/config/environment';
3 import { get } from '@ember/object';
4 import CheckSessionRoute from '.././check-session-route';
5 function service(user) {
6   return {
7     ...user,
8     userToken: get('currentUser.accessKey'),
9     userSecret: get('currentUser.userSecret'),
10  };
11 }
12 const user = get('currentUser.user');
13 export default CheckSessionRoute.extend({
14   - currentUser: service(),
15   + currentUser: service(user),
16
17   model() {
18     ...
19   }
20
21   ...
22
23 });

```

Listing 1. Sliced program with missing function argument

We will use Listing 1, which shows a real bug and its fix for a JavaScript project on GitHub, as a running example. This type of bug is common in many languages and falls under *Same function more args* pattern as categorized by Karampatsis and Sutton [26]. Let us consider how a developer would go about fixing the bug here. After localizing the faulty line of code, they would try to determine the root cause of the bug. To do so, they would analyze how the variables or functions in the faulty line have been affected in the previous lines. The developer notices that the property `currentUser` is invoking a function call to `service`. After examining the function declaration, they notice that `service` expects a required `user` argument and that the buggy line is not passing any arguments. The developer identifies the cause of the fault and generates a fix by passing the missing argument, `user` declared in line 12, to the function call.

Having too large of a context, especially if the method/class enclosing the buggy line is huge or has many dependencies can cause significant overhead [30]. By limiting the scope to only lines relevant to the fault, the developer can avoid information overload during the process of debugging and repairing the program.

In Listing 1, the highlighted code indicates the portion relevant for the developer to understand and fix the bug in line 14. The original buggy file in this example contained 66 lines of code and by focusing only on the relevant lines, the attention decreases to 13 lines. In practice, these kinds of bugs are pertinent in much more complex settings containing many lines and functions with more arguments. Furthermore, if we were to follow the approach of some of the current deep learning models, and extract only the enclosed method or enclosed class of the buggy line, then we would lose meaningful context outside the enclosing scope like the `user` variable and `service` function that were necessary for the developer in generating the correct fix.

Our insight is that for context, instead of constraining the attention to the enclosing entities or arbitrary limitations on the number of tokens/nodes, we can constrain the code to those portions that are relevant to the bug and fix through the notion of program slicing.

### 3 APPROACH

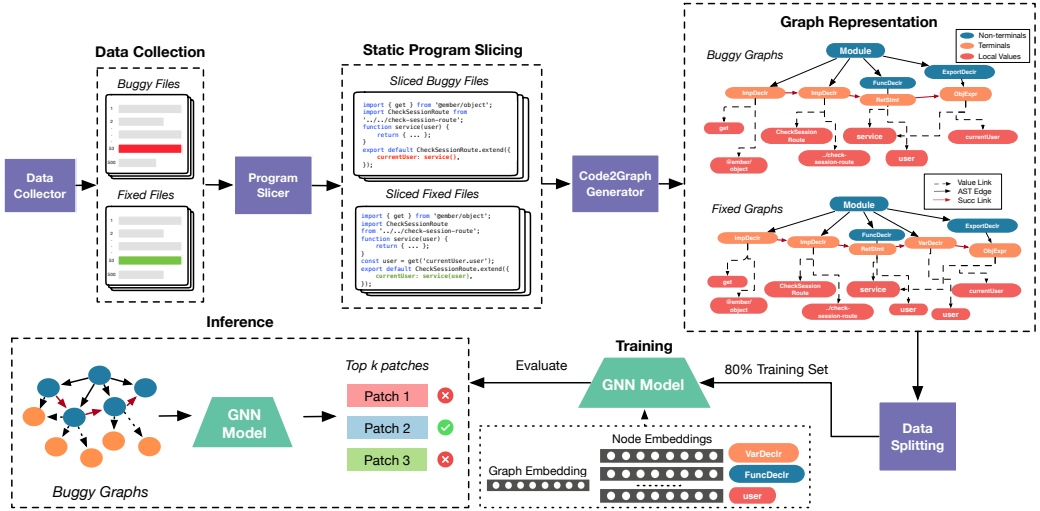


Fig. 1. Overview of our approach.

We hypothesize that slicing-based context can be helpful to a machine learning model that learns patterns of bug fixes. Our technique, called KATANA, is based on static program slicing to obtain relevant context with respect to the bug and its fix. This obtained slice-based context is then represented as an AST-based graph augmented with control and data flow edges. This graph is then fed into a Graph Neural Network (GNN) for training. Figure 1 depicts the overview of our approach. Our overall approach consists of five main steps, namely, data collection, program slicing, graph representation, training, and inference. Next, we describe each step.

#### 3.1 Data Collection

To collect data, we chose JavaScript as the target language. According to recent StackOverflow Developer Surveys [2], JavaScript stands as the most commonly used language in the world as reported by 68.62% of professional developers. JavaScript was initially limited to client-side code running in the web browser but has gained popularity in backend development through the emergence of the Node.js<sup>1</sup> framework. Unlike statically typed languages such as Java or C# where many errors can be discovered at compile time, JavaScript tends to "swallow" errors and silently continue execution. Therefore, having a good program repair tool for a dynamically typed language such as JavaScript can help developers with a seamless coding experience.

We collected pairs of buggy and fixed JavaScript files by crawling commits of open-source JavaScript repositories hosted on GitHub. We narrowed down the search by focusing on commits containing the keywords "bug", "fix" and "resolve". We collected data from top 1,135 JavaScript GitHub projects based on the number of stars. We exclude duplicates from the test dataset, as well as any data points that appear in both the test and training datasets [6]. Additionally, we use heuristics to filter out commits that may be feature additions or refactorings based on the number of AST differences. To this end, we consider a difference of one AST node to be a bug, similar to existing work [12]. A datapoint is considered to have one node difference if only a single AST node

<sup>1</sup><https://nodejs.org>

has been added, removed or modified in the buggy tree to obtain the fixed tree. This includes any internal node or leaf node e.g. operators, variables or values. Listing 2 demonstrates the examples of such cases. We extract these datapoints by 1) generating the AST of buggy and fixed files of each datapoint using SHIFT AST<sup>2</sup> parser and, 2) selecting only those datapoints where the diff between the two ASTs is equal to one. We exclude minified JavaScript files since our program analysis tool cannot handle those. We also prune any unparseable JavaScript datapoints using the SHIFT AST parser. If a JavaScript file has incomplete closures such as missing braces or parenthesis, we use the parser to check if an AST can be generated or not. We collected a total of 113,975 datapoints, i.e., pairs of buggy and fixed JavaScript files having one AST node difference between the pairs.

```

1 - sum(a, b);
2 + sum(a, b, c);
3
4 - return a - b;
5 + return a + b;

```

Listing 2. Example of one node difference

### 3.2 Static Program Slicing

We propose to use static analysis to examine the buggy line and all the ingredients, such as the variables and function calls used in that line, and backward program slicing [60], to determine where such ingredients have been defined, used, and modified throughout the code. We use this backward slice as context in our work, which is a subset of all statements with data or control dependencies with respect to the slicing criterion  $(p, V)$ , which is the set of variables  $V$  at the buggy program point  $p$ .

---

#### Algorithm 1 Extracting Backward Sliced Context

---

**Input:**  $N$ : A given line number;  $file$ : A given file

**Output:** List: String: List of statements

```

1:  $entitySet \leftarrow getEntities(N, file)$ 
2:  $contextLines \leftarrow \{N\}$ 
3: foreach  $entity \in entitySet$  do
4:   BACKWARDSLICE( $entity, contextLines$ )
5: end for
6: return  $getStatements(contextLines)$ 
7: function BACKWARDSLICE( $ent, contextLines$ )
8:   foreach  $ref \in ent.refs()$  do
9:     if  $ref.ent()$  is control-dependent or data-dependent on  $ent$  then
10:       $N \leftarrow ref.line()$ 
11:       $contextLines \leftarrow contextLines \cup N$ 
12:       $entitySet \leftarrow getEntities(N, file)$ 
13:      foreach  $entity \in entitySet$  do
14:        BACKWARDSLICE( $entity, contextLines$ )
15:      end for
16:     end if
17:   end for
18: end function

```

---

There are two types of program slicing analysis, namely, inter-procedural and intra-procedural analysis. Intra-procedural analysis is performed within the scope of the function and inter-procedural

<sup>2</sup><https://shift-ast.org>

analysis is performed within the whole program. We opted for performing both inter-procedural and intra-procedural analysis but limiting the slicing scope within the JavaScript file. The reason for such a choice is that only 12% (13,964) of the 113,975 datapoints have the buggy line enclosed within a function and 3% (22,940) have the buggy line as part of a class. In 85% (97,071), the buggy line is part of the global scope. In JavaScript, the global code can access all the constructs within a function, and the global code supports stepwise execution, just like functions do. This is unlike languages such as Java where the `main` method is required as the entry point for code execution. For this reason, we go beyond intra-procedural program analysis to capture a broad spectrum of datapoints. We kept our scope of analysis limited to the file itself, similar to other approaches [11, 12, 41, 57]. However, unlike these approaches, we do not apply any adhoc bounding limits and instead use backward slicing to capture repair ingredients systematically. This is because a limit based on an arbitrary number of tokens can potentially miss relevant context, particularly in cases where proximity to the contextual statements are above the bounding limit.

We analyze each datapoint in our dataset to collect the necessary information to conduct the slicing analysis. In order to determine the slicing criterion, we first perform a diff analysis between the buggy and fixed lines to obtain the buggy line number. We then feed this line number to our slicing framework and extract the variables and function calls used in that line. For extracting the contextual statements through static backward slicing, we incorporate both control flow and data flow analysis and identify the sliced statements where applicable.

Algorithm 1 illustrates a high-level description of the steps used in extracting the context from a given line number,  $N$  and  $file$ . Specifically, Algorithm 1 extracts the entities (i.e., variables, functions, objects) from the given buggy or fixed line in a file. Algorithm 1 then iterates through all the entities to obtain all the *contextLines* using backward slicing analysis. *contextLines* is a set of line numbers of the sliced context. In the *BackwardSlice* function, Algorithm 1 iterates through all the references in a given entity; if the referenced entity is control or data-dependent, then the corresponding line is added to *contextLines*; this process continues recursively to obtain the backward slice of the referenced entities. Finally, the algorithm constructs the source code statements from the *contextLines* to return the output. Here, the *getStatements* function ensures that all the closures of a given context line are extracted correctly, as it is a challenging process in a dynamic language such as JavaScript. For example, unlike Java, a semicolon is not required in JavaScript for declaration or expression statements; in such cases, we consider the new line as the end of a statement. The buggy line is within a loop or conditional statement in some cases. In such buggy statements, we determine the closure through control flow analysis to avoid an incomplete context with backward slicing.

```

1 import { get } from '@ember/object';
2 import CheckSessionRoute from '../../check-session-route';
3 function service(user) {
4   return {
5     ...user,
6     userToken: get('currentUser.accessKey'),
7     userSecret: get('currentUser.userSecret'),
8   };
9 }
10 export default CheckSessionRoute.extend({
11   -   currentUser: service(),
12 });

```

Listing 3. Sliced Buggy File

Our extracted datapoints of buggy and fixed files contain the whole JavaScript code with only the bug, and the patch that contains the difference at the same buggy line. We propose two types of slicing mechanisms, namely single slicing and dual slicing. For each, we generate a separate set

of datasets and conduct experiments to measure their effectiveness in repairing programs. These two types of slicing mechanisms are described below.

**Single Slicing.** Single slicing analysis is used for extracting context with respect to the buggy line. We start off by first taking the buggy line as the slicing criterion to capture the backward slice statements as context. This generates a buggy file with only the backward sliced statements as context. We then attach this context of the buggy file to the correct line and generate the fixed version of the buggy file. We call this *single slicing* because backward slicing has been applied only on the buggy file and the context of the buggy file is simply transferred to the fixed file. Listing 3 and 4 demonstrate the snippets of single sliced pairs of buggy and fixed files from the same example shown in Listing 1. As the slicing criterion was the buggy line, the fixed file contains the same sliced context of the buggy file and contains no information about the variable `user` that has been passed as a fix.

```

1 import { get } from '@ember/object';
2 import CheckSessionRoute from '../..../check-session-route';
3 function service(user) {
4   return {
5     ..user,
6     userToken: get('currentUser.accessKey'),
7     userSecret: get('currentUser.userSecret'),
8   };
9 }
10 export default CheckSessionRoute.extend({
11 +   currentUser: service(user),
12 });

```

Listing 4. Single Sliced Fixed File

```

1 import { get } from '@ember/object';
2 import CheckSessionRoute from '../..../check-session-route';
3 function service(user) {
4   return {
5     ..user,
6     userToken: get('currentUser.accessKey'),
7     userSecret: get('currentUser.userSecret'),
8   };
9 }
10 const user = get('currentUser.user'); // NEW CONTEXT
11 export default CheckSessionRoute.extend({
12 +   currentUser: service(user),
13 });

```

Listing 5. Dual Sliced Fixed File

**Dual Slicing.** Our intuition is that, certain repair ingredients are available in the fix context, which can improve the overall learning process. For instance, if the fix introduces a new identifier that was not present in the buggy line, then the corresponding context will be different in the fixed version of the file. This pattern is known as *Change Identifier Used*, and is commonly observed in many programming languages [25, 26]. Learning from the context of both the buggy and fixed code can provide additional semantic information to the model.

Therefore, we hypothesize that extracting slices from both the buggy and fixed files can be more effective in conveying the contextual information to the training model. We call this notion of collecting buggy and repair contextual ingredients *dual slicing*. Dual slicing analysis aims for extracting context separately from both buggy and fixed files. By using the slicing criterion from the buggy line and fixed line, we extract the sliced statements from both these lines individually to generate the sliced buggy and fixed files. Listing 3 and 5 demonstrate the dual sliced pairs of the buggy and fixed files from the motivating example of Listing 1. As the slicing criteria are both

the buggy and fixed lines in this approach, the fixed file contains an additional line containing the variable declaration of `user` in line 10 which is necessary for generating the patch as the function `service` expects this param.

### 3.3 Graph Representation

Once we have prepared the single sliced and dual sliced datapoints, we move to the learning phase of the approach. To represent source code, we opt for graph representation. Graph representations of source code have gained popularity recently due to their ability to represent semantic information [7, 12, 66, 68]. Typically, the program’s abstract syntax tree (AST) is used as the backbone of a program graph which consists of syntax non-terminals and terminals of a programming language’s grammar. Graphs are able to leverage both the syntactic and semantic relations between the nodes via different types of edges. They are also able to consider long-range dependencies through edges, between same variables or functions even if they are placed at distant locations [7]. This is unlike sequence based approaches which can sometimes lose meaningful contextual information due to a maximum token limit.

For each datapoint, we create graphs for the buggy and fixed sliced files separately. Following previous work [12], we extract the AST of each sliced files and convert it to a graph with the addition of `SuccToken` edges to connect leaf nodes and `ValueLink` edges to connect additional value nodes. Figure 1 illustrates the buggy and fixed graphs for the slices of our example (Listing 1). After representing the source code as a graph, the resulting graphs are mapped into a vector representation using a Graph Neural Network (GNN). More specifically, given a graph  $g = (V, E)$  with a set of nodes  $V$  and edges  $E$ , we determine the  $d$ -dimensional representation of graph  $g$  and individual node representations  $v \in V$  using a function  $f(g) \mapsto (\mathbb{R}^d, \mathbb{R}^{|V| \times d})$ . The node embedding is a vector,  $\vec{v} = h_v^{(L)}$  where  $L$  denotes the total number of propagations in the GNN via message passing [67]. Message passing is used in GNN models in which vector messages are exchanged between nodes in the graph and updated through an aggregation function. The graph representation  $\vec{g}$  is the aggregation of node embeddings  $h_v^l, \forall l \in 0, 1, \dots, L$ . To aggregate  $h_v^l$  for each  $l$ , max pooling is used, which takes the average of the  $L + 1$  vectors to obtain  $\vec{g}$ .

### 3.4 Training

In this step, we split the graphs generated from the previous step into training, testing and validation using random sampling. We fold the dataset into 80% training set, 10% testing set, and 10% validation set. Table 1 provides the number of datapoints in the training, validation, and test datasets in our study. To train a model, we adopt a GNN architecture that maps the graph representation into a fixed dimensional vector space. The program repair is essentially a series of graph transformation operations, from the buggy graph to the fixed graph, containing operations such as adding or deleting a node, replacing a node value or node type. During the graph creation step, the buggy graph ( $g_{bug}$ ), fixed graph ( $g_{fix}$ ) and the corresponding sequence of graph edits to create the  $g_{fix}$ , i.e., the graph diff is generated for each datapoint. In other words, the model creates  $g_{fix}$  by applying graph edit operations to the  $g_{bug}$ , on the buggy line. This graph diff comprises a sequence of AST modifications which serves as a fine-grained supervision mechanism for graph transformation. A modification contains the type of graph edit operation, node location in the graph and value to be added/modified (in case of deletion, no value is provided). To summarize, given a dataset  $D = \{(g_{bug}^{(i)}, g_{fix}^{(i)})\}_{i=1}^{|D|}$  containing pairs of buggy and fixed sliced code, the model is trained with the learning objective,  $\max_{\theta} \mathbb{E}_{(g_{bug}, g_{fix}) \sim DP}(g_{bug} | g_{fix}; \theta)$  to maximize the likelihood of fixes [12]. The Maximum Likelihood Estimation (MLE) objective is calculated as the sum of cross-entropy loss at each step of graph edits.



We train two separate models for each types of slicing. In single and dual slicing, the input and output to the model varies in terms of the contextual information. For single slicing, the input to the GNN model is the buggy graph,  $g_{bug}$  which contains the sliced context with respect to the buggy line only. The output of the model is a graph edit operation applied to the buggy node of  $g_{bug}$ , resulting in  $g_{fix}$ . On the other hand, for dual slicing, the model input is the buggy graph,  $g_{bug}$  which contains the sliced context with respect to the buggy line along with the fixed line. The output remains same as before i.e., a graph edit operation applied to the buggy node in  $g_{bug}$ . In short, the model trained with dual sliced context has access to extra contextual information pertaining to both the bug and fix. Note that, we do not differentiate the buggy or fixed graph input to the model based on any particular bug type, and feed the data as is.

Table 1. Dataset split for training, testing, and validation. The numbers for graph edit type are also indicated.

	ADD_NODE	DEL_NODE	REP_TYPE	REP_VAL	Total
Train	1,949	2,632	1,061	85,539	<b>91,181</b>
Validate	237	343	198	10,619	<b>11,397</b>
Test	243	370	209	10,575	<b>11,397</b>

**Hyperparameter Tuning.** Hyperparameters are important since they influence a model’s overall performance. We use a batch size of 10 and tune the hyperparameters of the model for the sliced datapoints. In order to find the optimum set of hyperparameters, we perform a manual search for discrete values of the number of GNN layers, learning rates, and dropout. To this end, we picked 2, 3 and 4 layers for the GNN model, learning rates of 0.1, 0.01, 0.001 and dropout values of 0.0, 0.1, and 0.2. We trained the model on 5 epochs with a batch size of 10 for each combination of hyperparameters. Overall, we trained 27 models during the hyperparameter search and found the best performance using 4 GNN layers, a learning rate of 0.001 and a dropout of 0.1.

### 3.5 Inference

After the training step, we evaluate each model using the test dataset. Given a buggy file during inference, we assume that the buggy line has already been identified in a fault localization step, similar to the current state-of-the-art learning-based repair techniques [11, 34, 41]. Locating the buggy line is a necessary pre-processing step to extract the backward slice as context. The model localizes the bug within the buggy graph and generates the corresponding patch. The input and output to the models during evaluation are as below:

- *Training:* As described in Section 3.4, we train two separate models with the single slicing and dual slicing datasets.
- *Inference:* Input to the models remains the same during inference. For evaluating the trained single and dual sliced models, input to the model is a graph,  $g_{bug}$  which is generated from the single-sliced buggy JavaScript file containing the buggy line.
- *Output:* For both cases, the model output is a series of graph edits applied to  $g_{bug}$  resulting into a fixed graph,  $g_{fix}$ . The model tries to generate  $T$  steps of transformations, where  $T$  denotes the number of graph edits. We set the value of  $T$  to 1 as our dataset contains bugs with one AST node difference.

For generating the fix for a  $g_{bug}$ , both single and dual slice models maintain a pool of 492 node types to predict the type; there are 5,001 values for the single slice model and 5,002 for the dual slice model based on the frequency of value tokens, in a global value dictionary to predict node

values; these types and value tokens are extracted from our training datasets. The extra vocabulary in the dual sliced model is `header-wrap`. It is the class name of a declared style used within React components. In this way, the vocabulary contains value tokens from both the buggy and fixed graphs, constructed from the buggy and fixed lines along with their sliced context. When generating patches to replace a node value, if a vocabulary is not in the global value dictionary, the patch is replaced with `UNKNOWN`. In the inference step, the model generates the top  $k$  patches for a buggy program as a series of graph edits depending on the beam size. During inference step for both single and dual slicing, the input to the model is a single sliced buggy graph  $g_{bug}$ . We consider an inferred patch as correct if the type, location and value of the fix has been accurately identified. This is achieved by matching the actual graph diff from the graph generation step with the inferred graph diff. For example, in case of Listing 1, the patch prediction is considered correct because the value ("user"), operation type (`ADD_NODE`) and AST node location (node index 44 in the AST) was correctly identified during inference.

### 3.6 Implementation

We implemented our technique in a tool called KATANA [4]. Since we were not able to find any existing program slicing tool that supports the latest JavaScript ES10 features, as required for analyzing our dataset, we implemented our own JavaScript slicer for KATANA. Following previous work [61], we build on top of the Understand [3] program analysis framework to analyze control flow and data flow dependencies in JavaScript code with respect to our slicing criterion. The slicer in KATANA takes as input a buggy JavaScript file, and the slicing criterion, which is the buggy line and the entities (variables, objects or functions) used in that line. It produces as output a sliced JavaScript file, which is used as context for learning repairs. The deep learning model in KATANA is based on the GNN architecture of HOPPITY [12].

## 4 EVALUATION

To assess the effectiveness of KATANA we address the following research questions:

- RQ1** How does dual slicing compare to single slicing for learning repairs?
- RQ2** How does KATANA compare to state-of-the-art learning-based repair techniques?
- RQ3** What is the effect of slicing on the information obtained as context?

We discuss the experiments that we designed to answer each of the research questions in the following subsections and outline the results.

### 4.1 Single versus dual slicing as context (RQ1)

We compare the effectiveness of single slicing where we extract context from the buggy file with dual slicing where the context is extracted from both the buggy and fixed files.

Using our initial 91,181 (training) datapoints, we generated two separate datasets using our single and dual slicing techniques. The processing time to generate slicing is negligible. On average, single slicing and dual-slicing analysis takes around 330 and 490 milliseconds for a single datapoint, respectively. With the tuned set of hyperparameters, we trained two models separately with the generated single sliced and dual sliced datasets. It took 1.7 hours to train each model for 50 epochs. The average inference time for a datapoint is around 2.54 seconds for each trained model. All our experimental runs are performed on an Ubuntu Linux 18.04.2 LTS server with 122 GB RAM, 16 vCPUs and 2,000 GB SSD.

We use the same test dataset to evaluate the accuracy of the two models. As the model produces a series of graph edits to generate the patch, we consider an output to be correct if the predicted location of the AST node, the type of operation (i.e., replace type or value, add node, delete node)

Table 2. Accuracy of single slicing vs. dual slicing

Approach	Accuracy		
	Top-1	Top-3	Top-5
Single Slicing	20.72%	35.01%	42.90%
Dual Slicing	28.31%	41.95%	46.96%

and the value of the node matches the expected node location, operation type, and value of the patch in the test dataset, for the entire sequence of graph edits.

Table 2 shows the results for single and dual slicing in KATANA. Single sliced context yielded accuracies of 20.72%, 35.01%, and 42.90% with beam sizes of one, three, and five, respectively. Dual sliced context yielded accuracies of 28.31%, 41.95%, and 46.96%, respectively. Dual slicing achieved higher accuracies for all beam sizes, with 37, 20, and 9 percent increases for the top-1, top-3, and top-5 suggestions, respectively, with respect to single slicing.

For instance, if we focus on top-3 suggestions, out of the 11,397 bugs in the test dataset, dual slicing fixed 4,781 and single slicing fixed 3,990. Out of these, 3,984 were fixed by both single and dual slicing, however, the dual slicing approach was able to fix 797 more bugs than single slicing.

```
- cart.push(object);
+ cart.push(item);
```

Listing 6. Example of *Change Identifier Used* bug pattern fixed by dual slicing

Listing 6 shows an example of a bug that was correctly fixed by dual slicing but was not repaired by single slicing. Through sampling, we verified that *Change Identifier Used* is one of the recurring patterns among the correct patches generated by dual slicing. We attribute this pattern to the availability of repair ingredients, made accessible through dual slicing as context used during training. Therefore, since the dual sliced context outperforms the single sliced context, we select it as our default approach in KATANA going forward.

## 4.2 Comparison with state-of-the-art (RQ2)

We compare KATANA with the four recent deep learning-based program repair techniques. We selected these state-of-the-art models based on the criteria that, (a) the technique supports JavaScript, (b) the artefact is available, and (c) if the technique does not support JavaScript, at the very least, it can be reimplemented and retrained for JavaScript. To this end, we compared KATANA with four state-of-the-art models. Among them, HOPPITY [12] and CoCoNuT [41] both support JavaScript and their artefacts are available for retraining a model from scratch using our dataset. On the other hand, Tufano et al. [57] and SEQUENCER [11] do not support JavaScript and were only trained on Java. However, these techniques are sequence based and do not rely on program analysis or language-specific features. Therefore, adding support for JavaScript was possible within their framework. In addition, we considered comparing KATANA to CURE [22], Recoder [69], DLFix [34], DEAR [35], and RewardRepair [64] using our dataset. Among these techniques, CURE, Recoder and RewardRepair rely on language-specific features and currently only support Java. While we investigated how to train DLFix and DEAR on our dataset, their artifacts were not available at the time of writing this manuscript (the authors confirmed in response to our emails).

A brief outline of these state-of-the-art deep learning-based program repair techniques is described below:

- (1) **Tufano et al.** [57] employs code abstraction on an RNN-based NMT model with attention mechanism.
- (2) **SEQUENCER** [11] uses an RNN-based NMT model equipped with copy mechanism.
- (3) **HOPPITY** [12] proposes a GNN model to predict the location of bug and generate a fix through a sequence of graph edits.
- (4) **CoCoNuT** [41] leverages a CNN-based NMT model with two separate encoders for buggy line and context.

4.2.1 *Characterizing context.* To characterize how current techniques treat context, we reason about five different aspects of context:

- **Analysis:** Contextual information can be extracted from the source code in various ways. The analysis can be as simple as naively extracting tokens from the buggy statement’s surroundings, to more complex program analysis techniques such as data flow, control flow, or slicing.
- **Representation:** Once extracted, context can be represented in different ways, e.g., as linear sequence of tokens, or nodes in a graph.
- **Scope:** Different levels of granularity can be considered for the scope of context, for example, by focusing on the entire program, enclosing file, class, or method.
- **Proximity:** Context can be extracted with respect to the location of the bug/fix, e.g., surrounding, before, or after the buggy line.
- **Limit:** Context size can be limited by the amount of information it contains, e.g., number of lines of code, tokens, or AST nodes.

Table 3. Accuracy comparison of KATANA with other learning-based program repair techniques.

Approach	Context					Accuracy		
	Analysis	Representation	Scope	Proximity	Limit	Top-1	Top-3	Top-5
Tufano et al. [57]	Naive	Sequences of tokens	Enclosing method	Before/After	100 tokens	4.90%	11.33%	15.84%
SEQUENCER [11]	Naive	Sequences of tokens	Enclosing class	Before/After	1000 tokens	7.99%	13.84%	18.08%
HOPPITY [12]	Naive	AST-based graph	Enclosing file	Before/After	500 nodes	5.05%	14.40%	19.62%
CoCoNuT [41]	Naive	Sequences of tokens	Enclosing method	Before/After	1,022 tokens	24.67%	27.90%	28.97%
<b>KATANA</b>	<b>Dual Slice</b>	<b>AST-based graph</b>	<b>Enclosing file</b>	<b>Before</b>	<b>N/A</b>	<b>28.31%</b>	<b>41.95%</b>	<b>46.96%</b>

Table 3 compares how state-of-the-art techniques treat context using these five aspects.

4.2.2 *Setup.* We trained a separate model using the deep learning framework of each techniques following the dataset split of Table 1 with our unsliced dataset as follows:

Tufano et al. [57] represent source code as a sequence of tokens and use the method enclosing the bug as the scope of context, which is limited by a maximum of 100 tokens. There is no program analysis for extracting context. However, abstraction is applied to code to limit the vocabulary size. We applied the same abstraction technique to our JavaScript dataset. We tokenized the buggy and fixed JavaScript files and tokenized it using js-tokens<sup>3</sup> to discern whether a given token is an identifier, method, or a literal. Following the same approach as [57], we maintain a dictionary of frequently occurring tokens and replace the remaining tokens with abstraction depending on the token type (e.g., METHOD\_1, VARIABLE\_1). Additional challenges with JavaScript code included the presence of JSX identifiers and literals. We assigned new abstractions for these types of tokens. The final vocabulary size became 815, including abstractions, JavaScript keywords, and frequent tokens.

<sup>3</sup><https://www.npmjs.com/package/js-tokens>

SEQUENCER represents source code as a sequence of tokens. However, they use the enclosing class as context scope, limited by 1,000 tokens; it delineates the buggy line within <START\_BUG> and <END\_BUG> tokens for differentiation within the context. SEQUENCER takes more input tokens from lines that appear before the buggy line. It extracts 2/3rd more tokens from preceding statements before the buggy line than the subsequent statements. We applied the same technique to our JavaScript code corpus, and if the buggy line was not encapsulated in a class, we extracted tokens from the surrounding lines in the JavaScript file. The vocabulary size is limited to 1,000 tokens.

HOPPITY represents context as an AST-based graph and does not apply any program analysis. Its context scope is the enclosing file of the buggy line, limited to 500 AST nodes. The vocabulary size on our dataset is 5,003 tokens.

CoCoNuT leverages the enclosed method of the buggy line as the scope of context and represents the source code as a sequence of tokens limited by 1,022 tokens. Similar to other techniques, context is taken without any program analysis. They employ abstraction and use two separate encoders to feed the buggy line and context. We used pre-processing scripts from their artefact to tokenize source code, and the resulting vocabulary size was 63,499.

We use the same test dataset of 11,397 JavaScript bugs for evaluating all the models. The best reported hyperparameter settings and epochs were used to train each of these models. To keep the accuracy comparison equitable across all the models, we use beam sizes of one, three, and five.

**4.2.3 Results.** Table 3 reports the obtained accuracies for beam sizes of one, three, and five in the columns *Top-1*, *Top-3* and *Top-5*. The lowest *Top-1* accuracy was yielded by Tufano et al. [57]. Despite using a high level of abstraction and a small vocabulary size, limited contextual information has not helped their model. As more contextual information was fed into the model, accuracy improved gradually. HOPPITY demonstrated better accuracy for *Top-3* and *Top-5* predictions. Among existing techniques, CoCoNuT yielded the highest accuracy. However, KATANA outperforms all existing techniques for all the beam sizes, with accuracies of 28.31% , 41.95%, and 46.96% for the beam width 1, 3, and 5, respectively.

Considering the top-3, KATANA is 270.26%, 203.11%, 191.32%, 50.36% more accurate in fixing buggy programs than Tufano et al. [57], SEQUENCER, HOPPITY, and CoCoNuT, respectively. By learning from the context of the relevant statements in the buggy and fixed file, the dual slicing-based context in KATANA can accurately repair buggy programs by a significant margin.

Figure 2 demonstrates an illustration of the overlapping sets of top-3 correct patches generated by the repair techniques. Each row in this figure represents a specific approach which is color encoded for differentiability. The numbers on the right depict the total number of patches generated correctly by each approach. For example, KATANA can fix a total of 4,781 out of 11,397 bugs, CoCoNuT can fix 3,180 out of 11,397 bugs correctly and so on. The numbers at the bottom indicate the intersecting set of the correct patches for the specific block. For example, 888 patches of the same set of bugs were correctly generated by KATANA, CoCoNuT, and SEQUENCER, as shown at the bottom. As three approaches overlap, this number is reflected at the top row in this column of 888 patches. All five techniques could accurately fix 101 instances of bugs as indicated by the presence of all colors in the leftmost first column in Figure 2. We observe that KATANA can fix a wide range of bugs, and the row for KATANA encompasses patches from other approaches. Furthermore, 891 bugs could only be fixed by KATANA as shown in the figure.

### 4.3 Slicing-based context (RQ3)

We carry out a quantitative analysis of our slicing technique to measure the type and amount of information considered as context and compare it against other approaches.

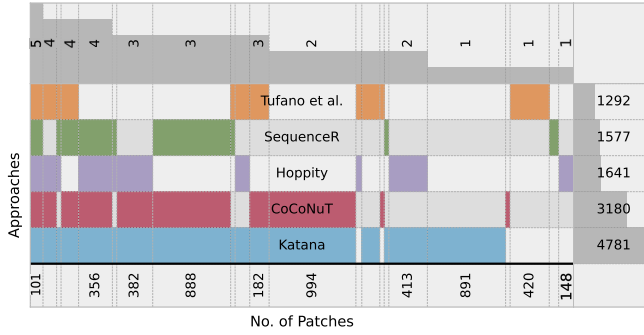


Fig. 2. Overlap of top-3 correct patches generated by context-aware learning based approaches.

We collected statistics over our dataset during the program slicing analysis. We calculated the number of lines before and after slicing as well as the fraction of the datapoints undergoing control flow and data flow analysis. We found that 26.96% of the total datapoints (113,975 pairs of buggy and fixed files) required the use of both control flow and data flow analysis to produce the slices. The remaining 73.04% were sliced by data flow analysis only. Figure 3 illustrates the number of lines before and after slicing across the datapoints. Here, we refer to the dataset before slicing as unsliced.

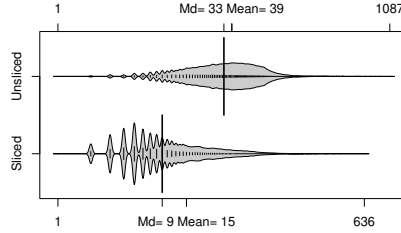


Fig. 3. Distribution of number of lines in Sliced and Unsliced data.

KATANA's program slicing reduced the average number of lines to consider for context from 39 to 15. The median number is also reduced from 33 to 9. The maximum number of lines before slicing is 1,087, and after slicing it is reduced to 636, whereas the minimum is 1 in both cases.

In Figure 4, we can observe the spread of line differences across the datapoints. The difference from sliced to unsliced is always positive indicating reduction after slicing. The average reduction is 24 lines. The maximum reduction is 1,073 lines, for a file that was 1,087 lines.

Figure 5 depicts bean plots of the number of tokens in the buggy files for KATANA and the other techniques. We chose the buggy files from our (113,975) JavaScript datapoints because the machine learning model takes the buggy file or buggy line with context as the input during inference. We stripped the whitespaces and comments from these files before token analysis. Tokens are the smallest possible unit that is common between all these approaches. Hence, we examine the increase or reduction of information used as context through this metric. As discussed before, except for KATANA, all of these context-based approaches use an ad-hoc bounding limit for reducing noise in the context. Just to recap, HOPPITY uses a maximum limit of 500 nodes whereas SEQUENCER, Tufano et al. and CoCoNuT use a sequence limit of 1,000, 100 and 1,022 tokens, respectively. Among all the approaches, HOPPITY has the highest maximum token count of 3,589 tokens followed by CoCoNuT (1,019 tokens), SEQUENCER (1,000 tokens) and Tufano et al. (100 tokens). The maximum number of

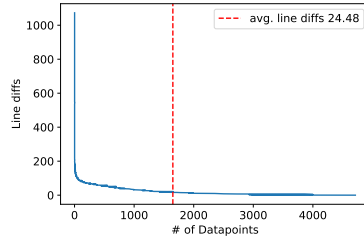


Fig. 4. Reduction in number of lines after slicing across datapoints.

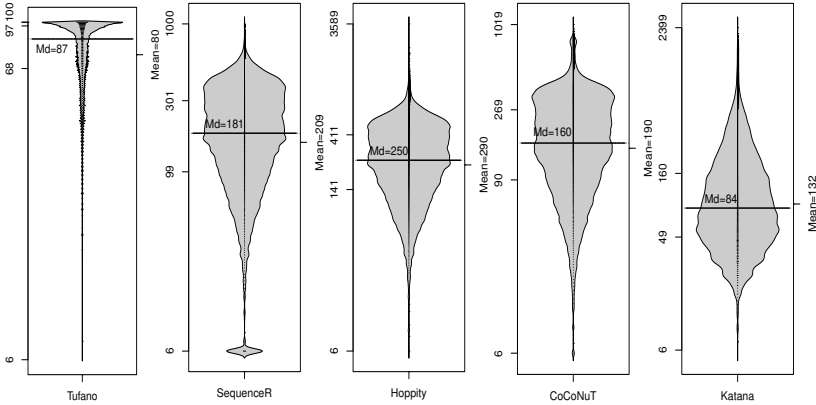


Fig. 5. Descriptive statistics of tokens in context-aware learning approaches.

tokens in KATANA is 2,399 tokens. Our slicing reduced the number of tokens by 1.5 times compared to HOPPITY. HOPPITY has the highest average number of 290 tokens, and with abstraction, Tufano et al. have the lowest average of 80 tokens. The average number of tokens in SEQUENCER and CoCoNuT are 209 and 190, respectively. KATANA has an average of 132 tokens, which is less than HOPPITY, SEQUENCER and CoCoNuT. The median number of tokens shows a more profound effect of program slicing in the reduction of tokens. KATANA has the lowest median of 84 tokens among all the approaches. We can also observe the same trend of token reduction in the lower quartile of the dataset with KATANA. For the third quartile of the dataset, HOPPITY, SEQUENCER and CoCoNuT have 411, 301, and 269 tokens, respectively. On the other hand, Tufano et al. and KATANA yielded 68 and 160 tokens, which is significantly lower. Without a need for truncating the context, our approach for extracting relevant context based on control and data flow analysis, has a significant amount of information reduction on average.

## 5 DISCUSSION

In this section, we discuss the findings from Section 4.

### 5.1 Role of slicing in learning repairs

In Table 3, we can observe that CoCoNuT outperforms other state-of-the-art program repair techniques. We believe that the reason for CoCoNuT outperforming the other techniques could be attributed to using contextual information from the enclosing method, adoption of a larger

model, and abstraction. CoCoNuT utilizes two separate encoders to encode the buggy lines and surrounding context. It uses abstraction to replace all strings and numbers (except 0 and 1) with `$STRING$` and `$NUMBER$` to avoid out-of-vocabulary tokens. This combination of abstraction, adopting a large model, and leveraging contextual information yields better accuracy for CoCoNuT. However, our approach KATANA outperforms CoCoNuT by a significant margin in the top-3 and top-5 accuracy. The overall accuracy of KATANA in repairing programs can mainly be attributed to the effectiveness of dual slicing-based context. Dual slicing can derive repair ingredients from both the buggy and fixed graphs. During training, the model generates a global value dictionary containing the top 5,000 most frequent tokens. In the inference step, the model leverages this vocabulary dictionary to predict the node value for graph edit operations e.g. `ADD_NODE` or `REP_VAL`. The model is able to find recurring patterns for node addition or replacement in the test set, despite having only the context of only the buggy line during inference. This is because the model has encountered similar cases of node addition or node value replacement in buggy graphs during the training phase. During single slicing, the model also captures tokens from buggy and fixed graphs. However, the context is same for both buggy and fixed graphs which affects the frequency of tokens in the vocabulary. Therefore, the global value dictionary does not contain any additional tokens that may have been possible if the context from fixed line was also considered. As demonstrated in Table 2, dual slicing is able to outperform single slicing by a significant margin due the extra contextual information. For instance, in Listing 6 the availability of vocabulary from the sliced context in fixed graphs during training equipped the model with repair ingredients needed for generating fixes during inference. Here, the fixed line replaces the variable `object` with `item` in which, the variable `item` is part of the vocabulary.

Listing 7 shows an example of a bug pattern that was fixed exclusively by KATANA. As shown, the fix for this bug was to add the argument `options` in the method call. In the sequence-based approaches, although the definition of variable `options` was present, the method definition of `generateInvite` was removed from the input sequence due to their maximum token limit. The reason is that the buggy file for this datapoint contains 435 lines of code which translates to 2,113 tokens. Although the method was defined within the enclosing class and enclosing file, the low proximity between the method call and method definition caused this method definition to be truncated. The method signature provides useful debugging information that maybe crucial to fix the bug. Through slicing, KATANA was able to retain relevant information pertaining to the buggy and fixed line. Moreover, the node value `options` is a commonly occurring token that was present in the global vocabulary dictionary. This indicates that dual slicing-based context can preserve repair ingredients, extract relevant information (e.g., method definition) while also reducing noise, which can benefit the learning process of the model, as shown in Table 3.

```
- this.generateInvite();
+ this.generateInvite(options);
```

Listing 7. Example of a bug fixed by KATANA

In addition to preserving the repair ingredients, program slicing can help to minimize noise for the GNN model. The quantitative analysis in Section 4.3 shows the effect of program slicing in reducing the information scope, without the need for a predefined bound on the number of tokens or AST nodes. Dual slicing allows the model to focus only on the relevant context while also reducing the distance with buggy line or patch.



Table 4. Qualitative analysis of bugs fixed exclusively by KATANA.

Bug Pattern	Description	Graph Edit	Buggy Code	Fixed Code
Same Function More Args	<i>Function with missing argument in the buggy line</i>	ADD_NODE	<code>return</code> drones[droneUpdate.id] = <b>new</b> Drone(droneUpdate); currentUser: service();	<code>return</code> drones[droneUpdate.id] = <b>new</b> Drone(droneUpdate, 10); currentUser: service(user);
Same Function Less Args	<i>Function with additional argument in the buggy line</i>	DEL_NODE	kittens.pop(name)  app.get('/api/current_user', (req, res, next)=> {	kittens.pop()  app.get('/api/current_user', (req, res)=> {
Incorrect Object Instantiation	<i>Constructor invoked without new keyword</i>	REP_TYPE	<b>var</b> componentSchema = Schema({  <b>const</b> history = mongoose.Schema({	<b>var</b> componentSchema = <b>new</b> Schema({  <b>const</b> history = <b>new</b> mongoose.Schema({
Missing Return Statement	<i>Expression with missing return keyword</i>	REP_TYPE	<b>await</b> bcrypt.compare(password, savedPassword);  changeAnimal();	<b>return await</b> bcrypt.compare(password, savedPassword);  <b>return</b> changeAnimal();
Incorrect Variable Declaration	<i>Variable declared with incorrect keyword</i>	REP_TYPE	<b>var</b> shuffled = arr.slice(0), i = arr.length, min = i - count, temp, index;  <b>var</b> garray = array//. <b>shift</b> ()	<b>let</b> shuffled = arr.slice(0), i = arr.length, min = i - count, temp, index;  <b>const</b> garray = array//. <b>shift</b> ()
Incorrect For Loop Used	<i>Expression with incorrect usage of for...in or for...of used</i>	REP_TYPE	for ( <b>var</b> file <b>in</b> files){  for ( <b>let</b> slot <b>of</b> Object.values(character.slots)){	for ( <b>var</b> file <b>of</b> files){  for ( <b>let</b> slot <b>in</b> Object.values(character.slots)){
Change Boolean Literal	<i>Expression with incorrect boolean literal used</i>	REP_VAL	app.use(bodyParser.urlencoded({extended: <b>false</b> }));  overlap: {type: <b>Boolean</b> , default: <b>true</b> },	app.use(bodyParser.urlencoded({extended: <b>true</b> }));  overlap: {type: <b>Boolean</b> , default: <b>false</b> },
Change String/Numeric Literal	<i>Expression with incorrect string/numeric literal used</i>	REP_VAL	<b>const</b> PORT = process.env.PORT    3001;  router.get('/comment/:id', CommentController.GetComments);	<b>const</b> PORT = process.env.PORT    3000;  router.get('/comment/list', CommentController.GetComments);
Change Identifier Used	<i>Expression with incorrect identifier used</i>	REP_VAL	componentWillMount: ()=> {  <b>this</b> .destroy(res, res)	componentDidMount: ()=> {  <b>this</b> .destroy(res, err)

## 5.2 KATANA is effective across pervasive bug patterns

Table 4 shows a qualitative analysis of the type and description of bug patterns observed in the test dataset that only KATANA was able to fix. Unlike prior techniques [33, 49, 53] which are trained on specific error types, our model targets a wide variety of error types as it is trained on real world bugs. We assessed 100 random samples out of the 891 correct patches generated uniquely by KATANA (see Figure 2, second column from right). As mentioned previously, our dataset has been curated from open-source GitHub repositories and hence, they are representative of real bugs that have been fixed by developers. For the qualitative analysis of the patches, the authors manually analyzed the resulting correct patches of KATANA, categorized them individually and came to a consensus if they agreed on the bug fixes to be representative of bug patterns. The assessment of the patches and categorization took approximately 12 person-hours in total. For labelling the bug type, we used ManySStuBs4J [26] to identify the category of the bug fixes. As this bug type categorization is a manual process and unfeasible to perform for the entire test dataset comprising of 11,397 datapoints, we only annotated a random sample of bugs with correct predictions by KATANA. However, Table 1 demonstrates the graph edit type for the overall dataset, which has been detected automatically. We present nine bug patterns (two patches per pattern as examples) fixed by KATANA in Table 4 in the test dataset. Among these nine bug patterns, five represent some common single statement bugs seen in other programming languages such as Java or Python [25, 26]. The remaining four patterns are specific to JavaScript. These language-specific bug patterns include *Incorrect Object Instantiation*, *Missing Return Statement*, *Incorrect Variable Declaration* and *Incorrect For Loop Used*. Since JavaScript code is interpreted at runtime, bugs such as *Incorrect Object Instantiation* and *Missing Return Statement* may only surface during program execution. These bugs occur because JavaScript is a dynamically typed language and can silently propagate these errors. *Missing Return Statement* has been categorized as a common bug pattern in JavaScript in BugsJS [18]. *Incorrect Variable Instantiation* is a common best practice in modern JavaScript projects that supports the ECMAScript 6 (ES6) syntax, and it indicates a code smell. In ES6, the keywords `let` and `const` were introduced to the language [1] because of scoping issues caused by the keyword `var`, which can eventually lead to hard-to-find bugs in JavaScript [55]. Another bug type, *Incorrect For Loop Used*, occurs mainly in ES6 compatible JavaScript source code, where the `for...in` and `for...of` loops are incorrectly used in which the former is used for looping through enumerables in objects and the latter is used for iterating through values of arrays and objects. Due to their syntactical similarity, developers often tend to use them interchangeably for the wrong purpose which can lead to bugs that are hard to identify. As shown in Table 4, the bug types *Same function more args* or *Same function less args* use the graph edit operations `ADD_NODE` and `DEL_NODE`, respectively. In such cases during inference, the model only generates the patch for the buggy line by adding/deleting the identifier or literal. In case node addition, if the given node exists in the vocabulary, then the model is able to identify the node element addition. Although the model was trained on dual sliced context from both buggy and fixed files, which can sometimes introduce new context or eliminate unnecessary context pertaining to the fix, it only localizes the buggy line and generates a corresponding patch during inference. Such instances of fixed graphs with varying context, during training, does not mislead the model and as such, it is able to identify these recurring patterns to find only the buggy line and generate patch. The most recurring pattern of bugs that KATANA could fix is *Change Identifier Used* which uses the graph edit operation `REP_VAL` (as exhibited in Listing 6 and Table 4). Compared to other state-of-the-art tools, KATANA is most effective in fixing *Change Identifier Used* bugs followed by *Same function more args* and *Same function less args*. This is because the extra vocabulary tokens obtained from the fixed graphs during training has equipped the model with the ability to identify and repair

these types of bugs. Thus, KATANA can detect and fix a wide variety of pervasive bug patterns that include both common and language-specific bugs more effectively than the baselines.

### 5.3 Effect of missing repair ingredients on inference

KATANA is most effective when it can incorporate relevant repair ingredients by leveraging context using slicing. To understand why KATANA was unable to correctly generate patches for some bugs, we manually examined 200 random samples from the incorrect patches. Upon careful investigation, we observed that for 16% of the incorrect patches, the relevant repair ingredients were not present within the file. Since we collect our datapoints based on buggy commits, our scope is limited to the enclosing file, which often does not include interdependent files. As such, we found instances of function calls or variables used in the buggy file that were imported from other files. As a result, relevant syntactic and semantic information required for such functions or variables was not preserved as context for the model to generate correct patches. To mitigate this, we plan to extend our slicing technique to perform interprocedural analysis *across files* as part of future work.

Furthermore, there were recurring instances (84%) of the *Change String/Numeric Literal* bug pattern among the incorrect patches generated by KATANA. Upon closer inspection, we noticed that not all of these string replacements were actual bug fixes—most of them were typo fixes or ad-hoc textual changes in config files. We believe that bug patterns pertaining to string changes can further benefit from a larger vocabulary size than the current 5,002 tokens.

```
- return React.createElement(STRING_6, {className: STRING_7})
+ return React.createElement(STRING_6, {className: STRING_9})
```

Listing 8. Example of a bug exclusively fixed by Tufano et al. [57].

However, as shown in Figure 2, KATANA was not able to fix 420 instances of bugs that Tufano et al. [57] alone would fix. Through random sampling, we identified most of the bug patterns to be string/literal changes among these 420 bug instances. The approach used by Tufano et al. [57] uses a renaming based abstraction [46] to limit the vocabulary size. The reason for this type of abstraction is that NMT models are not as effective in learning meaningful translations using a large vocabulary size; therefore, their approach applies abstraction to replace non-frequent identifiers and string/numeric literals with reusable tokens in the form of CATEGORY\_#. For example, STRING\_1, NUMBER\_1, VARIABLE\_1 etc. denotes the category (identifier/literal) and the number represents the sequential occurrence of that category within the code. Due to this abstraction, the search space during beam search decoding is significantly narrower in comparison to KATANA. For instance, Listing 8 depicts an example of a bug among the 420 exclusive correct patches by Tufano et al. [57]. As shown, the buggy string is replaced with the correct string which is present in the vocabulary of their model. However, the same string is not available in the vocabulary of KATANA as it is a non-frequent token. Therefore, KATANA is unable to predict the correct patch for such cases.

```
- return "foo";
+ return "bar";
```

Listing 9. Example of an adhoc string literal change bug

Dual slicing by KATANA is not as effective for certain cases of bug types *Change String/Numeric Literal*. These string/numeric literal changes (such as Listing 9) are merely typo fixes, or ad-hoc textual changes. For instance, in Section 4.1 (RQ1), we notice that although dual slicing fixed 797 more bugs than single slicing, six bugs were fixed exclusively by single slicing. After assessing these six bugs, we identified that they belong to the *Change String/Numeric Literal* pattern and were ad-hoc string changes. While dual slicing was able to predict the graph edit operation for

these bugs (`rep_val`), the value of changed string was not predicted correctly. To extract context, we use the immediate enclosing parent (i.e., file, method or class) as context, particularly when there are no variables or function calls in that line. Unless the changed string/number is present in the vocabulary dictionary, contextual information does not play an important role in program repair. Another scenario where dual slicing is not as effective is when the repair ingredients are not available within the file. This happens when an imported variable/function is used in the buggy line. For example, in Listing 7, we show that KATANA was able to fix the bug as the function `generateInvite` was declared within the file scope, however, if it were an imported function, then the model would not have any information about the method signature (i.e., how many parameters it expects). An interprocedural analysis that spans across files could be effective in capturing this additional context.

Currently, we capture the context of all variables/functions used in the buggy line. In some cases, context of one or more variables within a buggy line may be unnecessary to fix the bug. To handle such cases, one might potentially employ code instrumentation to capture relevant information from execution traces. However, steps to perform code instrumentation and extracting runtime traces could be potentially expensive. The reason is that it requires setting up the whole project, resolving the dependencies, and making the project compilable before executing the concerned buggy line. Furthermore, for a learning-based model to be effective, a large dataset of many such examples would be required. We instead rely on contextual information that could be derived statically. As part of future work, we plan to explore code instrumentation and dynamic analysis to curate a more accurate slice.

#### 5.4 Limitations

While KATANA is a research prototype, it could be envisioned as a plugin in a developer's IDE to facilitate program repair. For a given buggy code snippet, program slicing takes approximately 330 milliseconds. After the slicing step, the average execution time to generate a patch is 2.54 seconds, which makes it feasible to employ KATANA as a neural developer assistance tool.

To productize, KATANA could be extended by adding support for JavaScript files that contain templated code (i.e., HTML or JSX<sup>4</sup>) from front-end frameworks such as ReactJS<sup>5</sup>, VueJS<sup>6</sup> or Angular<sup>7</sup>. The current implementation of our analysis tool cannot extract slices for bugs inside front-end templated code. Consequently, KATANA could produce invalid code slices because the files do not contain pure JavaScript code.

KATANA cannot currently generate slices from minified or obfuscated JavaScript code because of constraints in our slicing framework. The minification step removes whitespaces, obfuscates variable names and often produces a single line as an output. Therefore, we excluded minified JavaScript files in our dataset as they may introduce noise into the learning process. However, in practice, this minification step is taken prior to deployment in the production environment, and it is not for human consumption. As a result, minified or obfuscated code would not limit the applicability of KATANA.

JavaScript is a dynamically typed language, which inherently makes it difficult to keep track of control and data flow dependencies during program slicing. In cases where control and data flow analysis does not yield any sliced context, we consider the contents of entire file as context scope for the buggy line.

---

<sup>4</sup><https://reactjs.org/docs/introducing-jsx.html>

<sup>5</sup><https://reactjs.org>

<sup>6</sup><https://vuejs.org>

<sup>7</sup><https://angular.io>

The current implementation of KATANA does not support multi-line patch generation. Since our dataset contains only one AST node difference, we use a single line as the slicing criterion to extract contextual statements. However, conceptually, we could extend the notion of program slicing for multi-line bug fixes.

We used both inter-procedural and intra-procedural backward slicing but limited the program analysis within the scope of a JavaScript file. One reason for such a choice is that our data collection is based on commits instead of projects; this approach cannot ensure that all the dependent files are present within the commit. Furthermore, because our underlying slicing framework, Understand [3], only supports static analysis within the file, our JavaScript slicer is constrained by this limitation. However, according to a previous work [9], a significant portion of the fixing ingredients can be found in the file containing the bug. As discussed in Section 5.3, the ability of KATANA to fix bugs is constrained by the availability of vocabulary. If a bug fix requires replacing a node or adding a node, the node value needs to be present in the vocabulary dictionary. Increasing the vocabulary size could potentially improve the performance of KATANA in fixing bugs.

We used backward slicing analysis, which may have resulted in the omission of some relevant contextual statements during aliasing. Aliasing is a phenomenon where more than one variable refers to the same location in memory. Certain bugs can occur when updating an alias for a variable, which in turn mutates the original variable. As these bugs can occur beyond the buggy line, forward slicing can help capture the relevant contextual statements. Support for forward slicing could be incorporated into KATANA for this bug pattern in future work.

## 6 THREATS TO VALIDITY

This section describes how we addressed potential limitations that may have biased our findings.

### 6.1 Internal Validity

An internal threat to validity is the accurate end-to-end replication of the comparison with state of the art techniques. Two of the techniques we compared with, namely Tufano et al. [57], and SEQUENCER applied custom abstraction mechanism on the enclosing method or class of bugs in Java source code. Following Tufano et al. [57], we implemented the same abstraction technique for JavaScript. SEQUENCER derives contextual information from the enclosed class of the buggy line. However, JavaScript poses unique differences from a compiled language such as, Java. As mentioned in Section 3.2, the buggy line in 85% of the dataset is part of the global scope. Since JavaScript source code is not always enclosed in a class or function, we take the entire file as the context scope in such cases to ensure that the same datapoints are preserved for a fair comparison across all techniques. The length of the buggy or fixed source code for these techniques are bounded by the maximum number of tokens that each of these techniques support; the context limit for Tufano et al. is 100 tokens, CoCoNuT is 1,022 tokens and SEQUENCER is 1,000 tokens.

Both Tufano et al. [57] and SEQUENCER are sequence based learning techniques where source code is treated as a stream of tokens for buggy Java files; we therefore used their replication package and pre-processed our data by adding support for JavaScript. CoCoNuT and HOPPITY already supported JavaScript implementation, hence we mainly trained these models on our dataset. Another threat to validity is the varied vocabulary size for each of the state-of-the-art techniques. We used the best reported hyperparameters and vocabulary size for the respective tools to ensure a fair comparison.

### 6.2 External Validity

We build our dataset from open source GitHub repositories of JavaScript projects. We used specific keywords to filter buggy commits following prior work [26]. One external validity is that the search heuristics used cannot ensure that all the datapoints represent actual bug fixes as some commits

may contain refactoring activity or ad-hoc code changes. Nevertheless, we were able to find nine types of bug patterns pervasive across the test set, and this search criteria is consistent with other learning-based program repair studies [12, 41, 57].

We implemented our approach for JavaScript, and our experiments cannot conclude how effective KATANA would be for other languages. However, the fundamental challenges KATANA addresses, i.e., how to retain useful contextual information for a repair task, why context selection using a naive approach is not sufficient, how to select useful information pertaining to the buggy/fixed line, are language independent and other languages could potentially benefit from slicing-based context. Static program slicing requires a slicing framework for each programming language, and building such a framework is an expensive process as it requires high development cost. Furthermore, state-of-the-art program repair techniques such as DLFix [34], HOPPITY [12], TFix [10] and SEQUENCER [11] also support a single language (i.e., Java or JavaScript) to evaluate the effectiveness of their approach. We therefore used a single programming language and built a slicing framework to evaluate our technique.

We considered a difference of one AST node for extracting buggy commits, which is similar to existing work [12]. As a result, currently our approach can handle one-off errors. Existing learning-based program repair techniques [8, 11, 32, 56] do not support multi-hunk changes (multiple buggy and fixed statements). However, conceptually KATANA could be extended to extract context from multiple buggy and fixed statements to better encode the repair ingredients, although this requires further evaluation in the future. In such approaches, using an ad-hoc approach would yield very large contextual information, severely limiting the effectiveness of a learning model. Therefore, we believe that fine-grained context extraction using KATANA would be even more relevant for the multi-hunk changes.

In this study, we proposed a novel technique, that leverages program slicing and the context from both buggy line and patch during training to fix bugs. We compared our approach with state-of-the-art learning based program repair techniques using the same dataset for training, validation and testing. We also performed an ablation study to understand the effect of slicing by using the same model and representation. The underlying GNN model and code representation used by KATANA is the same as HOPPITY. We re-trained HOPPITY using our JavaScript dataset without slicing the buggy/fixed files. We tuned the hyper-parameters of the GNN model using our sliced dataset and ran the model with single sliced and dual sliced buggy/fixed files. Thus, we were able to determine the effect of slicing using this experimental setup.

Some techniques do not use contextual information at all and others consider context using a different approach. For instance, Rachet [20] only considers the buggy line without any context, whereas DLFix [34] selects the buggy subtree as context. Although we did not directly compare with these in our evaluation, DLFix [34] reported to outperform Rachet, and CoCoNuT [41] reported to outperform DLFix. Since KATANA is directly compared with CoCoNuT, we can deduce, through transitive closure, that KATANA outperforms DLFix and Rachet, although direct experiments are needed to verify this empirically. TFix [10] is categorized as a program repair tool for static errors [44] and relies on reports from ESLint<sup>8</sup>, a popular static analysis tool for JavaScript. By generating the error reports based on the buggy code, TFix then merges the report with the buggy line along with its surrounding two lines of context to query the text-to-text transformer model (T5). As this has a dependency on these error reports from static analysis tools as context, we could not compare this approach with KATANA.

---

<sup>8</sup><https://eslint.org>

**Reproducibility.** We have made our dataset, model, comparison framework, and KATANA’s implementation available [4] for reproducibility of the results. We further provide instructions for replicating our experimental setup.

## 7 RELATED WORK

In this section, we describe related work on (a) how slicing is used in traditional program repair, (b) the usage of contextual information in learning-based program repair, and (c) the use of slicing on different learning-based source code processing tasks.

### 7.1 Traditional Automated Program Repair

Automated program repair has received significant attention from the research community. Many APR techniques have been proposed that could be classified as search-based [31, 32, 56, 58, 59], semantics-based [27, 47], or pattern-based [21, 28, 38–40, 52]. From this large body of literature, CapGen [61] is the closest to our work, which employs program slicing for program repair. CapGen is a search-based program repair technique that leverages context based on forward and backward slicing of nodes in the AST for mutation operator selection and ingredient prioritization. However, CapGen requires domain-specific knowledge about bug and fix types, whereas our approach can learn patterns from sliced data and fix a variety of bug patterns.

Additionally, there are repair techniques that address specific classes of faults [15]. For example, fault specific techniques focus on repairing conditional statements [13, 48, 63], concurrency bugs [23, 24], string sanitization [5, 65], access control violations [54], or repair memory leaks [14]. FixMeUp [54] targets access control violations in PHP web applications. FixMeUp applied interprocedural program slicing on the data, and control dependence graphs to identify the statements that need to be guarded by an access control check. Instead, we applied program slicing for a learning-based model which is not bug specific.

### 7.2 Learning-Based Program Repair

Most current learning-based repair techniques consider context in an ad-hoc manner. HOPPTTY [12] is limited by a maximum of 500 nodes in the AST and treats the whole file as context for the buggy line. On the other hand, SEQUENCER [11] uses the enclosing method and the class surrounding the buggy line to capture long-range dependencies between the buggy line and context. This technique is limited to 1,000 tokens, which may result in the truncation of code if the enclosing method is long, as an example. Tufano et al. [57] use buggy and fix files of small or medium-sized methods with a maximum of 50/100 tokens to learn bug fixes. They employ abstraction in the context of buggy and fixes files to mitigate the vocabulary limit. DLFix [34] is a deep learning-based program repair approach that uses the context of the surrounding subtree of both the buggy line and fixed line to generate fixes using a tree-based RNN model. This context is summarized as a vector and used as weights to generate the patch for the buggy code. CoCoNuT [41] uses the entire method of the buggy line as context with the buggy line and context fed as two separate inputs. In a following work, CURE [22] leverages GPT to provide contextual embedding for CoCoNuT. Recently, a pre-trained language model, CodeBERT [42], has been applied to fix Java simple bugs. There have also been attempts to generate patches in a refined way [69], generate multi-statement fixes [35] and to adjust the loss function during training iterations [64]. TFix [10] is a learning-based system that uses a T5 model [50] fine-tuned on a text-to-text patch prediction task that leverages the surrounding two lines of the buggy line as context for linting errors in JavaScript. All these techniques leverage context in a limited way and rely on predefined heuristics. In contrast, KATANA relies on program analysis to retrieve relevant code as context, is not bounded by predefined limits

or heuristics, and leverages control and data flow information from both the buggy and fixed versions of the code.

### 7.3 Slicing in Learning-Based Source Code Processing

Program slicing was explored in a learning-based vulnerability detection task [36, 37, 70]. VulDeePecker [37] trains a neural network with positive and negative examples to determine whether a code snippet suffers from vulnerability. A more recent development is  $\mu$ VulDeePecker [70], which extends VulDeePecker to predict multiclass vulnerabilities. These techniques extract slices of the vulnerable library/API function calls and are trained on an RNN to predict vulnerabilities in code. In a recent work, Xiao et al. [62] applied program slicing to a learning-based cryptographic API Suggestion. This approach applied interprocedural backward slicing from the invocation statement of a cryptography API and trained a modified LSTM-based model for a neural-network-based API recommendation task. In KATANA, we instead (a) apply program slicing on a generative task, i.e., fix generation, (b) introduce the notion of dual slicing to learn from both the buggy and fixed-versions of the code, and (c) employ a GNN model to better capture rich structural and semantic information that is inherent to the source code instead of using RNNs that treat code as a linear stream of tokens. To the best of our knowledge, we are the first to employ slicing for learning-based program repair.

## 8 CONCLUSION

As a developer, context is pivotal while understanding and writing a fix for a buggy piece of code. Surrounding lines, methods, and class level information provide background contextual information that is important to devise a patch for a buggy piece of code. Current learning-based repair techniques adopt predefined heuristics, such as a limited number of tokens or AST nodes to extract context, which is insufficient for representing contextual information of a bug fix. We argue that context extraction needs to be directed toward relevant code for a learning-based repair technique to be effective. We present KATANA, the first technique to apply program slicing on a learning-based program repair task that includes relevant information as context pertaining to the buggy/fixed code. We show that program slicing through control and data flow analysis effectively preserves sufficient program repair ingredients to extract recurring patterns. We propose the notion of *dual slicing*, a novel approach that leverages context from both the buggy and corrected code. Our approach is able to fix 4,781 out of 11,397 bugs curated from open-source JavaScript projects. The results show that KATANA can resolve between 1.5 up to 3.7 times more bugs when compared to the state-of-the-art learning-based repair techniques. Furthermore, KATANA is effective across a wide range of bug patterns. In the future, we plan to expand our interprocedural analysis, which is now limited to the enclosing file. We will also expand KATANA to support other programming languages such as Java.

## REFERENCES

- [1] 2015. ECMAScript 2015 Language Specification - ECMA-262 6th Edition. <https://262.ecma-international.org/6.0/>. Accessed: 2022-01-07.
- [2] 2021. StackOverflow Developer Survey 2021. <https://insights.stackoverflow.com/survey/2021/#most-popular-technologies-language-prof>. Accessed: 2022-01-26.
- [3] 2021. Understand by Scitools. <https://www.scitools.com/>. Accessed: 2021-12-30.
- [4] 2022. Katana. <https://github.com/saltlab/Katana>. Accessed: 2022-11-25.
- [5] Muath Alkhalaf, Abdulbaki Aydin, and Tevfik Bultan. 2014. Semantic Differential Repair for Input Validation and Sanitization. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014)*. Association for Computing Machinery, 225–236.
- [6] Miltiadis Allamanis. 2019. The Adverse Effects of Code Duplication in Machine Learning Models of Code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. Association for Computing Machinery, 143–153.



- [7] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations (ICLR)*. 520–524.
- [8] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proceedings of the ACM on Programming Languages OOPSLA (2019)*, 27 pages.
- [9] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The Plastic Surgery Hypothesis (FSE '14). Association for Computing Machinery, 306–317.
- [10] Berkay Berabi, Jingxuan He, Veselin Raychev, and Martin Vechev. 2021. Tfix: Learning to fix coding errors with a text-to-text transformer. In *International Conference on Machine Learning*. PMLR, 780–791.
- [11] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyanyk, and M. Monperrus. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering (2019)*, 1–1.
- [12] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *International Conference on Learning Representations (ICLR)*.
- [13] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proceedings of the 11th International Workshop on Automation of Software Test (Austin, Texas) (AST '16)*. Association for Computing Machinery, 85–91.
- [14] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-Leak Fixing for C Programs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, 459–470.
- [15] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* 45, 1 (2019), 34–67.
- [16] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. AAAI Press, 1345–1351.
- [17] Shivani Gupta and Atul Gupta. 2019. Dealing with Noise Problem in Machine Learning Data-sets: A Systematic Review. *Procedia Computer Science (2019)*, 466–474.
- [18] Péter Gyimesi, Béla Vancsics, Andrea Stocco, Davood Mazinanian, Árpád Beszédes, Rudolf Ferenc, and Ali Mesbah. 2019. BugsJS: a Benchmark of JavaScript Bugs. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 90–101.
- [19] Sakib Haque, Alexander LeClair, Lingfei Wu, and Collin McMillan. 2020. *Improved Automatic Summarization of Subroutines via Attention to File Context*. Association for Computing Machinery, 300–310.
- [20] Hideaki Hata, Emad Shihab, and Graham Neubig. 2019. Learning to Generate Corrective Patches using Neural Machine Translation. *arXiv preprint arXiv:1812.07170 (2019)*.
- [21] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, 298–309.
- [22] N. Jiang, T. Lutellier, and L. Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 1161–1173.
- [23] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-Violation Fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, 389–400.
- [24] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated Concurrency-Bug Fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, 221–236.
- [25] Arthur V. Kamienski, Luisa Palechor, Cor-Paul Bezemer, and Abram Hindle. 2021. PySStuBs: Characterizing Single-Statement Bugs in Popular Open-Source Python Projects. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 520–524.
- [26] Rafael-Michael Karampatsis and Charles Sutton. 2020. *How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset*. Association for Computing Machinery, 573–577.
- [27] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press, 295–306.
- [28] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. IEEE Press, 802–811.
- [29] Jindae Kim and Sunghun Kim. 2019. Automatic patch generation with context-based change application. *Empirical Software Engineering (2019)*, 4071–4106.

- [30] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Softw. Eng.* 32, 12 (dec 2006), 971–987.
- [31] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for \$8 Each. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*. IEEE Press, 3–13.
- [32] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38 (2012), 54–72.
- [33] Guangjie Li, Hui Liu, Jiahao Jin, and Qasim Umer. 2020. Deep Learning Based Identification of Suspicious RETURN Statements. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 480–491.
- [34] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. Association for Computing Machinery, 602–614.
- [35] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A Novel Deep Learning-Based Approach for Automated Program Repair. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, 511–523.
- [36] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* (2021), 1–1.
- [37] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. *Proceedings of the Symposium on Network and Distributed System Security* (2018).
- [38] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 456–467.
- [39] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. *TBar: Revisiting Template-Based Automated Program Repair*. Association for Computing Machinery, 31–42.
- [40] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 118–129.
- [41] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 101–114.
- [42] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying CodeBERT for Automated Program Repair of Java Simple Bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. 505–509.
- [43] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: Learning to Repair Compilation Errors. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. Association for Computing Machinery, 925–936.
- [44] Martin Monperrus. 2018. *The Living Review on Automated Program Repair*. Technical Report. HAL Archives Ouvertes. <https://hal.archives-ouvertes.fr/hal-01956501>
- [45] Simonetta Montemagni and Vito Pirrelli. 1998. Augmenting WordNet-like lexical resources with distributional evidence. An application-oriented perspective. *Proceedings of the COLING/ACL Workshop on Use of WordNet in Natural Language Processing Systems*, 87–93.
- [46] Marjane Namavar, Noor Nashid, and Ali Mesbah. 2021. A Controlled Experiment of Different Code Representations for Learning-Based Bug Repair. <https://doi.org/10.48550/ARXIV.2110.14081>
- [47] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. 772–781.
- [48] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (1996), 99–118.
- [49] Michael Pradel and Koushik Sen. 2018. DeepBugs: A Learning Approach to Name-Based Bug Detection. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 25 pages.
- [50] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research* 21, 140 (2020), 1–67.
- [51] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspán. 2018. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* 61, 4 (2018), 58–66.
- [52] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 648–659.

- [53] Roger Scott, Joseph Ranieri, Lucja Kot, and Vineeth Kashyap. 2020. Out of Sight, Out of Place: Detecting and Assessing Swapped Arguments. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 227–237.
- [54] Soeul Son, Vitaly Shmatikov, and Kathryn S McKinley. 2013. FixMeUp: Repairing Access-Control Bugs in Web Applications. In *Network and Distributed System Security Symposium (NDSS)* (network and distributed system security symposium (ndss) ed.).
- [55] Luis Sotomayor. 2022. Avoiding JavaScript Scoping Pitfalls. <https://nearsoft.com/blog/avoiding-javascript-scoping-pitfalls>. Accessed: 2022-01-06.
- [56] Shin Hwei Tan and Abhik Roychoudhury. 2015. Relifix: Automated Repair of Software Regressions. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE)*. 471–482.
- [57] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Transactions on Software Engineering and Methodology* (2019), 29 pages.
- [58] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 356–366.
- [59] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 364–374.
- [60] Mark D. Weiser. 1979. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*. 439–449.
- [61] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 1–11.
- [62] Ya Xiao, Salman Ahmed, Wenjia Song, Xinyang Ge, Bimal Viswanath, and Danfeng Yao. 2021. Embedding Code Contexts for Cryptographic API Suggestion: New Methodologies and Comparisons. *arXiv preprint arXiv:2103.08747* (2021).
- [63] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Softw. Eng.* 43, 1 (2017), 34–55.
- [64] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-Based Backpropagation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. Association for Computing Machinery, 1506–1518.
- [65] Fang Yu, Muath Alkhalaf, and Tefrik Bultan. 2011. Patching Vulnerabilities with Sanitization Synthesis. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. Association for Computing Machinery, 251–260.
- [66] Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. 2022. Learning to Represent Programs with Heterogeneous Graphs. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. 378–389.
- [67] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* (2020), 57–81.
- [68] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. Curran Associates Inc.
- [69] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, 341–353.
- [70] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019.  $\mu$ VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* (2019), 1–1.