Context-Aware Code Change Embedding for Better Patch Correctness Assessment

BO LIN and SHANGWEN WANG, National University of Defense Technology, Changsha, China
MING WEN, Huazhong University of Science and Technology, Wuhan, China
XIAOGUANG MAO, National University of Defense Technology, Changsha, China

Despite the capability in successfully fixing more and more real-world bugs, existing **Automated Program Repair (APR)** techniques are still challenged by the long-standing overfitting problem (i.e., a generated patch that passes all tests is actually incorrect). Plenty of approaches have been proposed for **automated patch correctness assessment (APCA**). Nonetheless, dynamic ones (i.e., those that needed to execute tests) are time-consuming while static ones (i.e., those built on top of static code features) are less precise. Therefore, embedding techniques have been proposed recently, which assess patch correctness via embedding token sequences extracted from the changed code of a generated patch. However, existing techniques rarely considered the context information and program structures of a generated patch, which are crucial for patch correctness assessment as revealed by existing studies. In this study, we explore the idea of context-aware code change embedding considering program structures for patch correctness assessment. Specifically, given a patch, we not only focus on the changed code but also take the correlated unchanged part into consideration, through which the context information can be extracted and leveraged. We then utilize the *AST path* technique for representation where the structure information from AST node can be captured. Finally, based on several pre-defined heuristics, we build a deep learning based classifier to predict the correctness of the patch. We implemented this idea as Cache and performed extensive experiments to assess its effectiveness. Our results demonstrate that Cache can (1) perform better than previous representation learning based techniques (e.g., Cache relatively outperforms existing techniques by ≈6%, ≈3%, and ≈16%, respectively under three diverse experiment settings), and (2) achieve overall higher performance than existing APCA techniques while even being more precise than certain dynamic ones including PATCH-SIM (92.9% vs. 83.0%). Further results reveal that the context information and program structures leveraged by Cache contributed significantly to its outstanding performance.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; *Software defect analysis*; Software testing and debugging;

Additional Key Words and Phrases: Automated program repair, patch correctness, deep learning

This work is supported by the National Natural Science Foundation of China No. 61872445, No. 62002125, and No. 61672529.
Authors' addresses: B. Lin, S. Wang (corresponding author), and X. Mao, National University of Defense Technology, Changsha, China; emails: {linbo19, wangshangwen13, xgmao}@nudt.edu.cn; M. Wen (corresponding author), Huazhong University of Science and Technology, Wuhan, China; email: mwenaa@hust.edu.cn.

1049-331X/2022/05-ART51 $15.00
https://doi.org/10.1145/3505247

ACM Reference format:
Bo Lin, Shangwen Wang, Ming Wen, and Xiaoguang Mao. 2022. Context-Aware Code Change Embedding for Better Patch Correctness Assessment. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 51 (May 2022), 29 pages.
https://doi.org/10.1145/3505247

1 INTRODUCTION

Automated Program Repair (APR) has attracted huge attention over recent years, and plenty of techniques have been proposed. Existing APR techniques are generating an increasing number of promising results, thus promoting software automation to a large extent [38, 40, 41, 44, 63, 68, 78].

Nonetheless, due to the lack of strong program specifications (e.g., formal descriptions of the desired behavior), APR techniques often leverage the developer-provided test suites as the criteria to assess patch correctness. However, such a strategy, unfortunately, leads to the long-standing challenge of APR: the *overfitting* problem [36, 39, 55, 64, 71, 79, 94, 96]. For instance, a patch simply deleting the intended function can be generated and deemed as a correct one if the inadequate test suite does not examine this intended function [55]. In other words, a patch generated by APR techniques that passes all the tests may still be incorrect. Such a phenomenon is devastating when applying APR techniques in practice considering the situation in which tremendous time and efforts have been consumed while the generated patch is still buggy or even has removed intended functionalities.

Over the years, many approaches have been proposed to assess patch correctness (i.e., to determine whether a generated patch is indeed correct or not) [74, 76, 85, 87, 91–93]. These techniques can be generally classified into two categories: those that need to run tests (i.e., *dynamic*) and those built on top of static code patterns or features (i.e., *static*). Dynamic ones directly perform assessments based on whether the extra generated tests from automated test generation tools (e.g., Randoop [65] or EvoSuite [23]) are passed [60, 93] or the run-time information further collected [85, 87]. In contrast, static techniques focus on code change patterns [74], or code similarities between the buggy and patched code with respect to their syntax and semantics [81]. Following a previous study [81], we denote these **Automated Patch Correctness Assessment** techniques as **APCA techniques** to ease our presentation. Despite promising results, existing APCA techniques all have their own weaknesses. For instance, dynamic ones are extremely time-consuming [85, 93] and thus are unwieldy. Static ones, although being more efficient, are less precise [81].

With certain large-scale patch benchmarks being released recently [16, 30, 51], researchers propose to leverage representation learning techniques (e.g., BERT [15] and CC2Vec [26]) to generate code embedding for assessing patch correctness [76], taking advantage of the power of big data. Despite promising results achieved by existing representation learning based techniques, their effectiveness is compromised by two major limitations. Specifically, (1) they only focus on the buggy and patched statements while ignoring the surrounding *context information* (i.e., other correlated statements within the method), and (2) they simply treat code as token sequences, thus missing *program structures* such as the types of statements. However, the context information and program structures are of significant importance to assess patches' correctness as revealed by existing studies [13, 47, 57].

In this article, we present a deep learning based APCA approach called **Cache**, which stands for **C̲ontext-a̲ware code c̲h̲ange e̲mbedding**, to assess the correctness of generated patches. Cache is context-aware since, given a code change (i.e., a patch), it does not only focus on the changed statements but also takes the other correlated statements within the same method (i.e., the unchanged part) into consideration. By doing so, knowledge such as the types and values of the variables in the changed statements can be leveraged by Cache. The identification of the changed and unchanged code is achieved via differentiating the **Abstract Syntax Trees (ASTs)** of the buggy and patched methods. Cache is also aware of the program structures since it adopts the *AST path* [5] representation of code for embeddings. Through representing programs as *AST path*, the inherent program structure information of the AST node can be captured by Cache and thus helps it better examine the code change. To our best knowledge, incorporating context information and program structure is new for code change embedding as well as patch correctness assessment.

To train and evaluate Cache, we construct two patch datasets based on existing benchmarks, a smaller one containing a total of 1,183 patches and a larger one containing a total of 50,794 patches. Our basic idea is to include as many existing patches as possible. In the small one, we focus purely on patches from the Defects4J benchmark [29] in that most of the existing APR tools were evaluated on it. We combine two existing large-scale benchmarks from Wang et al. [81] and Tian et al. [76] after a filtering process of deduplication. In the large one, with the aim to assess the generality of Cache, we include patches from the RepairThemAll framework [16] which were generated over diverse defect benchmarks. To build a balanced dataset, we also include patches from ManySStuBs4J [30] which contains substantial correct patches extracted from diverse open source projects. This dataset is constructed after deduplication and removal of test code. Results reveal that Cache significantly outperforms previous representation learning techniques and also achieves the optimum performance among all the APCA techniques. Specifically, the F1-score achieved by Cache is 78% on the small dataset, while the optimum performance achieved by existing representation learning techniques is 73.7% obtained by CC2Vec [26] with the Logistic Regression classifier. Additionally, Cache is more precise than several dynamic APCA techniques, such as PATCH-SIM [87]: Cache achieves the precision of 92.9%, while 83.0% is achieved by PATCH-SIM on the dataset from Wang et al. [81]. Moreover, an ablation study shows that the context information and program structures can significantly contribute to the performance of Cache. In summary, our study makes the following contributions:

- We propose a novel context-aware code change embedding technique. To the best of our knowledge, this is the first approach that incorporates the context information in embedding code changes.
- We apply the embedding approach in the domain of patch correctness assessment and implement a tool named Cache. The Cache is open-source to facilitate future APR studies at: **https://github.com/Ringbo/Cache**.
- We conduct various experiments to evaluate Cache on diverse patch benchmarks. The results show that Cache achieves overall significantly better performance than not only previous representation learning techniques but also existing APCA techniques.

2 BACKGROUND & MOTIVATION

2.1 Representation Learning

Learning distributed representations has attracted huge attention in recent years with numerous embedding approaches emerging, targeting both natural languages and source code [2, 5, 15, 98]. Doc2Vec [35] is an unsupervised algorithm used to learn fixed-length feature representations from sentences, paragraphs, and documents with variable-length pieces of texts. Similarly, code2vec [5] and code2seq [2] are attention-based neural models for representing code snippets as a single fixed-length code vector by training on AST paths (i.e., the sequences of AST nodes, which symbolize the parent-child relation between adjacent nodes in the tree) and code tokens. BERT [15] is a natural language representation model developed by Google, which is hammered at pre-trained deep bidirectional representations from unlabelled texts by jointly conditioning on bidirectional contexts in all layers. To embed code changes, CC2vec [26] encodes consecutively code tokens, statements, chunks, and files while Lozoya et al. [56] and Yin et al. [95] represent code changes using the syntactic structure information. However, none of them takes into consideration the context information (i.e., other correlated statements within the method), which will be shown to play a critical role in code change embedding in this study.

2.2 Automated Patch Correctness Assessment

Long et al. [55] first pointed out that test suites are inadequate for shaping the specifications of a program, resulting in a patch passing all the tests potentially still being incorrect. After that, various APCA techniques have been proposed for validating patch correctness. Recently, Wang et al. [81] performed a systematic study on the effectiveness of existing APCA techniques. They mainly observe that static heuristics based on code features are generally imprecise, while dynamic APCA techniques (i.e., those that needed to execute tests) can identify a smaller number of overfitting patches. We briefly review the included techniques in their study as we are going to compare against them in our evaluation. DiffTGen [85] assesses whether a patch is correct via comparing the outputs of the generated patch and the ground-truth on the same tests, a patch is considered as correct if its execution results on the tests are the same as those of the ground-truth. Anti-patterns [74] checks if the code change in a patch violates human-defined rules, a patch is considered as correct if it does not violate any of the rules. Note that this technique was originally designed for C language. To apply it on Java language, the common way in the community is to manually perform the check [81, 87]. PATCH-SIM [87] approximates the correctness of a patch based on the execution trace without the oracle information, a patch is considered as correct if its execution traces on passing/failing tests are similar/different to those of the original buggy program. Opad [91] first generates tests and then assesses patch correctness based on the criterion that a correct patch should not expose exceptions during test runs. Some researchers [93] propose to generate tests at a large scale using test generation tools such as Evosuite [23] and Randoop [65] for augmenting the test suite, and a patch is considered as correct if it can pass all the externally generated tests. Daikon [19], a tool for inferring program invariants, is recently utilized for assessing patch correctness with the idea that two programs are semantically equivalent if their inferred invariants are the same [90], a patch is thus considered as correct if its inferred invariants are identical to those of the ground-truth. Readers can refer to the previous work [81] for the details about the mechanisms of the existing APCA techniques. ODS [92] leverages manually-designed static features from patches and trains machine learning based classifiers for patch correctness. ODS also considers context information to define features that can describe the modified statements. Our work differs from it in that in Cache, the features in context are automatically learned to better represent code changes; while the features in ODS are immutable and ad hoc to predict patch correctness. As we will show in the following motivation section, manually-designed features cannot describe the code change comprehensively. In our evaluation (cf. Section 5.2), our Cache indeed outperforms ODS. Tian et al. [76] proposed to leverage code representation techniques for differentiating correct and overfitting patches and adopted four state-of-the-art techniques (i.e., BERT, CC2vec, code2vec, and Doc2vec) to demonstrate that it could achieve a promising performance compared with the state-of-the-art techniques such as PATCH-SIM. Our proposed Cache can also be considered as a representation learning technique. It is fully static while most of the existing APCA techniques studied by Wang et al. [81] are dynamic. We utilize the context information and the program structure to represent a patch (which can be considered as a code change) and then predict its correctness by a classifier. It is more reasonable than the existing representation learning techniques studied by Tian et al. [76] where patches are represented as code token sequences without the help from the context and program structure information. Note that Chen et al. [11] proposed UniAPR which is an on-the-fly patch validation approach. Their main objective is to reduce the time cost in the process of patch generation while ours is to help assess the correctness of those generated patches.

2.3 Motivation

Inrecent years, a number of large-scale patch benchmarks have been released [16, 30, 43, 51]. Enabled by the power of big data, the idea of leveraging representation learning techniques to identify correct patches is emerging. For instance, considering the example demonstrated in Figure 1, representation learning based techniques, such as CC2vec, would treat the added and deleted lines separately by considering the whole line as a token sequence and embedding each single token. In this case, it would encode the tokens `{ int, len, =, thisBuf, length, -, strLen }` in order and concatenate them for representing the buggy code, while encoding the tokens `{ int, len, =, size, -, strLen, +, 1 }` for representing the patched code. However, there are two main drawbacks of this strategy, notably. ❶ The context information (i.e., other correlated statements within the method) is not taken into consideration. Such information is of great importance since a previous study [82] has demonstrated that the frequency of code changes to fix bugs diverse significantly under different contexts with respect to AST node types. For instance, within the context of `Method Declaration`, the insertion of a `Expression Statement` occurs over ten times more frequently than the insertion of an `If Statement` from the statistics of a dataset containing over 3,000 real fixing patches [82]. In our example, the patch transfers a variable `thisBuf.length` into a new one `size`. Nonetheless, without the information of this variable, such as its type and value, it would be difficult to assess whether this code change is correct. By contrast, if we can take line 1 into consideration, we would obtain the knowledge that the variable `size` is with the same type as `strLen`. Further, if we take line 9 into consideration, we would know that the variable `strLen` is an integer. Such facts concerning the surrounding context of a patch can help us better understand the code change, thus facilitating us embedding the patch more precisely. ❷ The information of program structures has been missed. Current techniques consider the statement as a token sequence and embed each token separately. However, treating keywords in a program in the same way with other tokens (e.g., variable names) may overlook specific functionalities of tokens and thus overlook the program's inherent structure. For instance, keywords such as `if` and `for` represent the starting of a conditional statement or a loop body. Ignoring such facts is problematic since a recent study [47] has pointed out that the naturalness of faulty elements differs across various types of statements. That is to say, how often a repair action (e.g., add, delete, or replace) is adopted in a specific patch diverges significantly with respect to different statement types. Specifically, more than half of the assignment statements require code replacement to fix bugs, while the corresponding proportion of conditional statements is lower than 40%. Hence, if we know the type of statements where the bug resides, plus with the code change information performed on the statements, we can better examine the correctness of the code change. Due to the lack of context and structure information, existing representation learning techniques cannot identify the example as a correct patch. For example, BERT and Doc2Vec both mislabelled the ground-truth patch as an incorrect one [76].

To overcome the limitations as mentioned above, we propose Cache, a code change embedding approach that captures both the context and structure information. For problem ❶, we use *code diff* tools [47] to split the changed and unchanged parts within the buggy method where the whole unchanged part is utilized as the context. In the example as shown in Figure 1, we treat all other lines within this method except lines 5 and 6 as context. By doing so, information from line 1 and line 9 can be encoded to help us better assess the patch (i.e., changing `thisBuf.length` to `size`). It should be noted that ODS [92] also takes the context information (i.e., line 1) into consideration. Nonetheless, manually-designed features are insufficient to fully describe the context information. Consequently, it will only focus on the statement type of line 1 (i.e., `IfStatement`) while gaining

```
        if (strLen > size) {
            return -1;
        }
        char[] thisBuf = buffer;
-       int len = thisBuf.length - strLen;
+       int len = size - strLen + 1;
        outer:
        for (int i = startIndex; i < len; i++) {
            for (int j = 0; j < strLen; j++) {
```

Fig. 1. The ground-truth patch for bug Lang-61.

limited understanding about the type of the variable `size`, which plays critical roles in predicting the correctness of this patch as mentioned above. In contrast, our approach adopts a deep learning technique to represent the context information. The desired information from the context is therefore automatically extracted and learned. Theoretically, performing program slicing [24, 99] can help us identify relevant statements more precisely to the changed ones. It, however, is either computation-consuming (static slicing) or dependent on specific inputs (dynamic slicing), and thus is not included in our current approach since we need to run on a large-scale dataset and we do not rely on the test information. For problem ❷, we use AST paths [5] to embed patches which, to a large extent, preserves the structure of the original program since any two consecutive nodes in the AST path are inherent with the *parent & child* relation in the parsed AST. Details will be introduced in the next section.

3 PROPOSED APPROACH

In this section, we introduce our proposed Cache in detail. We first define the terminologies used in our approach and then explain each part of our designed approach.

3.1 Definition

We leverage the **Abstract Syntax Tree (AST)** for code representation. Specifically, we represent code as *AST path*. We first present the following definitions before introducing our approach. Please note that our definitions strictly follow the previous work code2vec [5].

3.1.1 Definition 1 - AST. The AST of a code snippet is defined as a tuple $\langle N, L, T, r, \Omega, \Psi \rangle$, where N is a set of non-leaf nodes; L is a set of leaf nodes; T is a set of code tokens for L; $r \in N$ represents the root node; Ω is a function $\Omega : N \longrightarrow (N \cup L)^{\star}$ that maps non-leaf nodes to their children; and Ψ is a function $\Psi : L \longrightarrow T$ that maps leaf nodes to their corresponding code tokens.

3.1.2 Definition 2 - AST path. Given an AST $\mathbf{A} = \langle N, L, T, r, \Omega, \Psi \rangle$, an AST path is defined in this study as a sequence of nodes $p = e_1, e_2, \ldots, e_k$ where $e_1, e_k \in L$, $e_2, \ldots, e_{k-1} \in N$, and for every consecutive pair of nodes e_i and e_{i+1}, there always holds $e_i \in \Omega(e_{i+1})$ or $e_{i+1} \in \Omega(e_i)$. Intuitively, an AST path is a path that connects two leaf nodes through a series of non-leaf nodes on the tree.

3.2 Approach Overview

The overall pipeline of Cache is illustrated in Figure 2. Generally speaking, our approach accepts a buggy method and a patched method as inputs. Cache then identifies the buggy sub-tree and its corresponding patched sub-tree. After identifying the buggy and patched sub-trees, the remaining part of the AST of the buggy method is considered as an individual AST and served as the context information in our approach. We adopt an AST path based code representation technique for embedding an AST since such a technique can capture both the program structure information

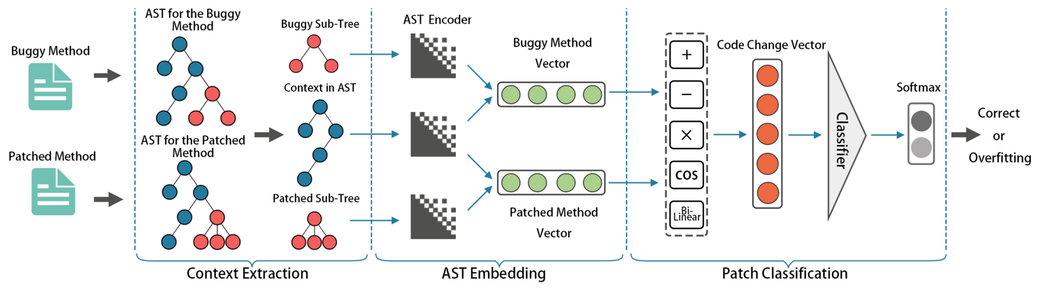


Fig. 2. The overall framework of Cache.

encoded by the path and the semantic information encoded by the leaf node. The AST path technique has shown its great capability on code representation [2, 5, 9]. After obtaining the representations for the buggy and patched methods, we use five pre-defined functions for integrating the representations and then use a deep learning based classifier for generating the final result. More details will be explained in the following. Note that to ease our presentation, we suppose the inputs of Cache only contains a pair of buggy and patched method. However, our approach can also handle those cases whose fixing patches reside across multiple methods (see Section 3.5).

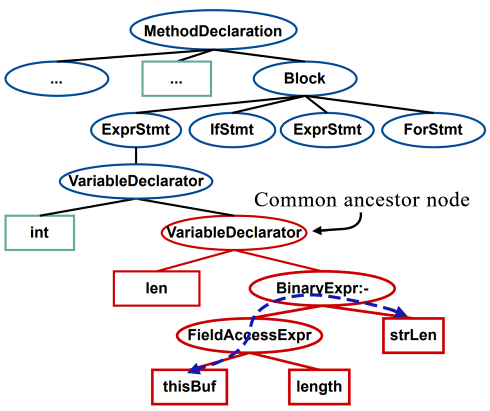
3.3 Context Extraction

Given a method pair (i.e., the buggy method M_b and the patched method M_p) as inputs, Cache generates the ASTs of these methods using a widely-adopted tool, Gumtree [21]. It then detects the common ancestor node (N) of the buggy sub-tree and patched sub-tree via locating the changed nodes in M_b (those have been updated or deleted) and the added nodes in M_p through *AST diffs* [47]. The AST diffs technique can calculate the edit actions on an AST. Therefore, by traversing the nodes marked with “change operation” in the returned AST, we can obtain all the changed nodes. The sub-tree within M_b whose root node is N is thus considered as the buggy sub-tree while the one within M_p with the same root node is treated as the patched sub-tree. After that, the remaining nodes of the AST of M_b (or AST of M_p) are considered as an individual tree which provides the context information. Therefore, for a given method pair, Cache in total generates three ASTs, which are a buggy sub-tree (namely A_b^{sub}), a patched sub-tree (namely A_p^{sub}), and a context AST (namely A_{Con}). It should be noted that Cache allows the fixes to occur in multiple places via identifying the common ancestor node (N) for these fixes.

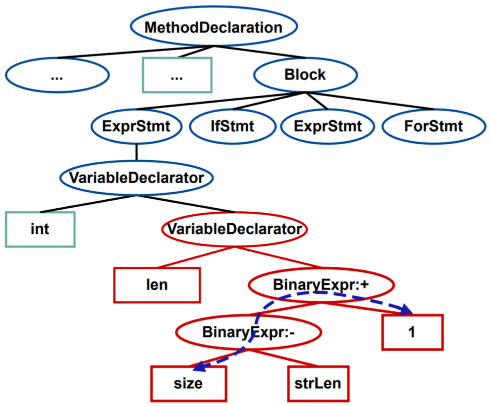
Figure 3 presents a concrete example which shows the ASTs of the buggy and patched programs as shown in Figure 1. Each leaf node is attached with its associated code token (in rectangles). For simplification, the other AST nodes irrelevant to the patch are not presented in the figure. Specifically, the patch modifies a `BinaryExpression` within a `VariableDeclaration` statement into another one. Cache first identifies the highlighted `VariableDeclarator` node as the common ancestor node. After that, the nodes depicted in blue orals are denoted as the A_{Con}, while the nodes in red orals are denoted as the A_b^{sub} (in the left subfigure) and A_p^{sub} (in the right subfigure). We also use blue dotted lines to demonstrate two AST paths in the buggy and patched sub-trees, respectively.

3.4 AST Embedding

3.4.1 AST Paths Extraction. Given A_b^{sub}, A_p^{sub}, and A_{Con}, we first extract all the AST paths for each of them. In this work, we use an off-the-shelf AST path collector [33] (i.e., *PathMiner*) which traverses the AST in a bottom-up manner and generates all paths for this AST. We choose to



(a) AST for buggy code snippet.



(b) AST for patched code snippet.

Fig. 3. The AST of the buggy and patched versions for the code snippet in Figure 1.

reuse this tool since it provides a framework to extract the AST paths effectively. Furthermore, it provides several external interfaces, such as the customized output which can facilitate our data collection. In our study, if two leaf nodes can be connected by several paths, we consider the one with the least number of AST node for representation with the aim to reduce the number of nodes in an AST path. The intuition is from the previous study [4] which shows that long paths may cause the representation space to be too sparse and thus reduce the generality of learned model. Additionally, we also collect code tokens of the leaf nodes in this step and include them as features for embedding. The intuition is that code tokens preserve semantic information from the original programs (e.g., the names of the variables) and it has been demonstrated that it can help improve the precision and recall values of models based on AST paths [2]. After this step, we obtain the AST paths and the corresponding code tokens of A_b^{sub}, A_p^{sub}, and A_{Con}.

3.4.2 *Model Architecture.* Figure 4 shows the overview of our encoding model. As for embedding an AST, we design a network based on the open source implementation of code2seq [2], which is mainly composed of the attention mechanism [77]. Given an AST path, we first encode the two tokens of the leaf nodes and the whole AST path to generate the representation for the AST path. Then, with the vector representations of all the AST paths in the original AST, we automatically learn the attention weight on each path through an attention model and generate the vector representation for the whole AST. The attention mechanism is used to select distributions over these AST paths dynamically.

Embedding AST path and tokens: In this step, we first split the tokens of the leaf nodes into sub-tokens [2, 46] according to the camel case and underscore naming conventions. The sub-tokens are embedded using a learned matrix, and the full token is represented by the sum of vectors of sub-tokens. We also learn another matrix for embedding the AST node based on the node type. The nodes in the AST path sequence are then sent into an LSTM encoder whose output is considered as the vector representation of the AST path. At last, the vectors of the two tokens and the AST path are concatenated for representing the AST path.

Formally, given a set of AST paths $\{p_1, p_2, \ldots, p_k\}$, our model embeds a vector representation $\mathbf{v}_i$ for each path p_i. For $p_i = \{e_1^i, e_2^i, \ldots, e_{l_i}^i\}$, t_{i1} and t_{i2} are code tokens of e_1^i and $e_{l_i}^i$, the embedding

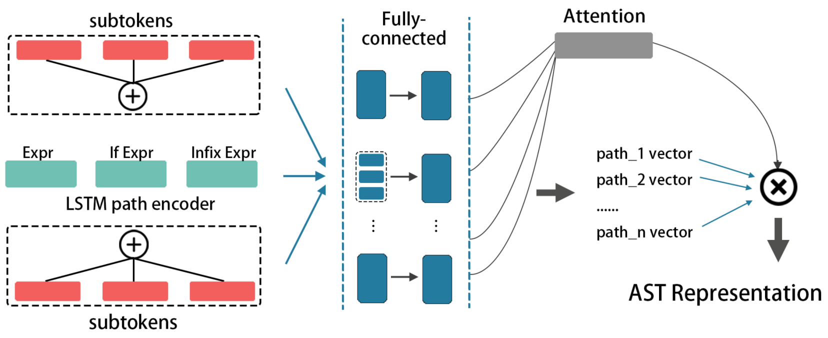


Fig. 4. The architecture of our AST embedding approach.

process is formulated as below:

$$\begin{aligned}
\mathbf{V}_{t_{i1}} &\leftarrow \sum_{t_s \in T_{i1}} \mathbf{E}_t(t_s) \\
\mathbf{V}_{t_{i2}} &\leftarrow \sum_{t_s \in T_{i2}} \mathbf{E}_t(t_s) \\
\mathbf{V}_{p_i} &\leftarrow LSTM(\mathbf{E}_p(e_1^i), \mathbf{E}_p(e_2^i), \ldots, \mathbf{E}_p(e_{l_i}^i))
\end{aligned} \tag{1}$$

where $\mathbf{E}_t(*)$ and $\mathbf{E}_p(*)$ are the embedding matrix for sub-tokens and AST node, T_{i1} and T_{i2} are sets of sub-tokens of t_{i1} and t_{i2}, $\mathbf{V}_{t_{i1}}$ and $\mathbf{V}_{t_{i2}}$ represent the vector representation of code tokens t_{i1} and t_{i2}, $LSTM$ denotes the bi-directional LSTM neural network, and $\mathbf{V}_{pi}$ is the output of the node sequence. Then the concatenation of $\mathbf{V}_{t_{i1}}$, $\mathbf{V}_{pi}$ and $\mathbf{V}_{t_{i2}}$, which is defined as $\mathbf{V}_i$, is used to represent the whole AST path p_i.

Extracting features with the attention network: In this step, given the representations of the AST paths in the tree, we pass them through a fully-connected layer and an attention network sequentially for generating the representation of the whole AST (namely $\mathbf{E}_{tree}$). The attention mechanism computes a weighted average over the combined vectors, and its main job is to compute a scalar weight for each of them [5].

Formally, given the representations of AST paths in a tree $\{\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_k\}$, the process can be formulated as:

$$\begin{aligned}
\mathbf{z}_i &\leftarrow tanh(\mathbf{W}_f \times \mathbf{V}_i) \\
a_i &\leftarrow \frac{exp(\mathbf{z}_i^T \cdot \mathbf{a})}{\sum_{j=1}^{n} exp(\mathbf{z}_j^T \cdot \mathbf{a})} \\
\mathbf{E}_{tree} &\leftarrow \sum_{i=1}^{k} a_i \cdot \mathbf{z}_i
\end{aligned} \tag{2}$$

where $\mathbf{W}_f$ is the weight matrix of the fully-connected layer, $\mathbf{a}$ is the attention vector which is initialized randomly and learned simultaneously with the network, a_i is the attention weight of z_i, which is computed as the normalized inner product between z_i and the global attention vector $\mathbf{a}$, and $\mathbf{E}_{tree}$ is a linear combination of vectors $\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_k\}$ factored by their attention weights to represent the whole AST.

We recall that the inputs of this step (i.e., AST embedding) are three ASTs: A_b^{sub}, A_p^{sub}, and A_{Con}, respectively. After using our model, we obtain three vector representations for each of them, which are E_{B_t}, E_{P_t}, and E_{Con}. We then concatenate E_{B_t} and E_{Con} for representing the buggy method (E_{B_m}) while concatenate E_{P_t} and E_{Con} for representing the patched method (E_{P_m}).

3.5 Patch Classification

We assess the correctness of a generated patch via comparing it against the original buggy patch. Therefore, given the vector representations of the buggy and patched methods, we first need to integrate those two vectors and then design a classifier for automatically assessing the patch correctness.

3.5.1 Integrating Representations. Given two vectors $\mathbf{E}_{B_m}$ and $\mathbf{E}_{P_m}$, both with n dimensions, we leverage five different functions to integrate them with the aim of characterizing the differences between $\mathbf{E}_{B_m}$ and $\mathbf{E}_{P_m}$ from diverse aspects. Such five functions are selected due to their promising results achieved as reported in previous studies [26, 75, 76].

① $\mathbf{E}_{add}$: **Element-wise addition**, which is an addition operation between $\mathbf{E}_{B_m}$ and $\mathbf{E}_{P_m}$.
② $\mathbf{E}_{sub}$: **Element-wise subtraction**, which is a subtraction operation between $\mathbf{E}_{B_m}$ and $\mathbf{E}_{P_m}$.
③ $\mathbf{E}_{mul}$: **Hadamard product**, which is a Hadamard product between $\mathbf{E}_{B_m}$ and $\mathbf{E}_{P_m}$.
④ $\mathbf{E}_{sim}$: **Cosine similarity**, which captures the similarity between $\mathbf{E}_{B_m}$ and $\mathbf{E}_{P_m}$ and is regarded as a one-dimensional vector since it is a real number.
⑤ $\mathbf{E}_{bi}$: **Bilinear model**, which is proposed by Tanenbaum et al. [75] and has been shown to be able to efficiently combine the features of two vectors.

We then concatenate the above five integrated vectors (denoted as $\mathbf{E}_a$) and then feed it as the input to our final classifier as follows:

$$\mathbf{E}_\mathrm{a} \leftarrow [\mathbf{E}_{add}; \mathbf{E}_{sub}; \mathbf{E}_{mul}; \mathbf{E}_{sim}; \mathbf{E}_{bi}] \tag{3}$$

where dimensions of $\mathbf{E}_{add}$, $\mathbf{E}_{sub}$, $\mathbf{E}_{mul}$, and $\mathbf{E}_{bi}$ are all n while that of $\mathbf{E}_{sim}$ is 1, leading to the dimension of $\mathbf{E}_a$ being $4n + 1$. In our study, the value of n is set to 320 via following the previous study code2seq [2] where a method is represented as a 320-dimentional vector.

If the generated patch modifies code elements across multiple methods, Cache will compute the $\mathbf{E}_\mathrm{a}$ for each of the methods, and then perform bit-wise addition to integrate those vectors together to represent the code change of this patch.

3.5.2 Classifier. Unlike a previous study [76] where widely-used machine learning models are utilized for determining the patch correctness, in our study, to extract more hidden code change features, we feed the embedding vector $\mathbf{E}_\mathrm{a}$ into a neural network which is composed of two fully-connected layers and a binary predictor. Finally, we apply a standard softmax function to obtain the probability distribution over correctness. A patch is labelled as correct if its probability of being correct ($p(\mathbf{E}_{\mathrm{out_c}})$) is larger than that of being incorrect ($p(\mathbf{E}_{\mathrm{out_i}})$); otherwise, it is considered as overfitting.

3.6 Training

To train the network, we use the cross-entropy loss [69], which has been widely used in multi-classification tasks [2, 5, 53, 83, 101]. That is, parameters in our model are learned via minimizing the following objective function:

$$L = \sum_i -[y_i \cdot log(p(\mathbf{E}_{\mathrm{out_c}})) + (1 - y_i) \cdot log(p(\mathbf{E}_{\mathrm{out_i}}))]$$

where $y_i = \{ 1,0 \}$ denotes whether the i_{th} patch is correct or overfitting. We employ the dropout technique [73] to improve the robustness of Cache and the Adam approach [31] to minimize the objective function.

4 STUDY DESIGN

4.1 Research Questions

- **[RQ-1]** *To what extent can CACHE achieve better performance against existing representation learning techniques?* We compare with five representation learning techniques (BERT [15], code2vec [5], code2seq [2], Doc2Vec [35], and CC2Vec [26]) to see to what extent CACHE outperforms the state-of-the-art.
- **[RQ-2]** *What is the performance of CACHE against existing dynamic APCA techniques and heuristics based on static code features?* We assess CACHE on the patch benchmark from a systematic study [81] to compare its effectiveness against both dynamic and static techniques. A latest machine learning based APCA technique, ODS [92], was also evaluated on this benchmark and is thus included in our study.
- **[RQ-3]** *To what extent do our design choices affect the overall effectiveness of CACHE?* We investigate the impact of diverse design decisions (e.g., whether considering the context information and program structures when embedding ASTs) on the effectiveness of CACHE.

4.2 Experiment Settings

Our CACHE was trained on a server with Ubuntu18.04x64 OS, 32GB memory, and NVIDIA 1660ti GPU. The following presents the experimental details of each research question.

4.2.1 [Dataset Collection]. With the developments of APR researches, many benchmarks containing both correct patches and incorrect ones have been created and released. Such datasets enable CACHE to learn precise embedding for patch correctness assessment. In this study, we construct two patch datasets based on existing patch benchmarks [16, 30, 76, 81], a smaller one containing 1,183 labeled patches and a larger one containing 50,794 patches.

In the small dataset, we focus purely on patches from the Defects4J benchmark [29] since most of the existing APR tools were evaluated on Defects4J. We need to include comprehensive patch benchmarks that are built on Defects4J. Specifically, we select patches from two existing large-scale datasets. One is created by Wang et al. [81] containing 902 patches and the other by Tian et al. [76] containing 1,000 patches. Patches in these two studies were either written by developers (i.e., the ground-truth patches) or generated by up to 22 different APR tools, and their correctness has been carefully labeled. Specifically, authors of the previous studies [76, 81] have manually checked the correctness of each included patch using the open rules provided by Liu et al. [51]. However, these two benchmarks were created independently, and thus their included patches might be duplicated. Since we are going to perform 5-fold cross-validation, the patch used for training may also occur in the testing set if there exists duplicated patches, which will cause data leakage. To avoid such issues, we construct the small dataset via combining these two benchmarks after a filtering process of deduplication, which ends in 1,183 patches. This process is performed as follows: since a patch is stored as a diff file, for each patch, we remove all the blank spaces in the file, and then compare the left text information with that from the other patches. If two patches are identical with respect to their text information, they are considered as duplicated.

Beyond Defects4J, other defect benchmarks have been proposed such as BEARS [59], Bugs.jar [70], IntroclassJava [18], and QuixBugs [45]. Hence, to investigate the generality of CACHE, we also include those patches generated on these benchmarks to construct a large patch dataset. Fortunately, the RepairThemAll framework [16] has generated substantial patches over these diverse defect benchmarks, and thus is selected in our study. Nevertheless, it has been shown that the majority of patches from this benchmark are overfitting ones [76] (the correctness of the patches is manually labeled by the authors). Moreover, it has been highlighted in recent studies [76, 81, 92] that the dataset should be balanced (i.e., the number of correct patches and that of overfitting

Table 1. Datasets Used in Our Experiments

Datasets	Subjects	# Correct patches	# Overfitting patches	Total
Small	Tian et al. [76]	468	532	1,000
	Wang et al. [81]	248	654	902
	Filtered	**535**	**648**	**1,183**
Large	ManySStuBs4J [30]	51,433	0	51,433
	RepairThemAll [16]	900	63,393	64,293†
	Filtered	**25,589**	**24,105**	**49,694**

†Patches from IntroclassJava benchmark are discarded since they are student programs, following the previous study [76].

ones should be similar) when performing learning-based experiments. As a result, we need to include a large number of correct patches into our large dataset. We decide to include patches from ManySStuBs4J [30] for compensating the lack of correct patches. Patches from this benchmark are mined from over 1,000 different open-source projects, which are genuine fixes of real-world defects. They are thus considered as correct patches in our large dataset. Note that we discard test code from ManySStuBs4J for avoiding potential biases and also perform deduplication on this large dataset. We note that the numbers of correct patches from RepairThemAll dataset before/after deduplication are 94/900, which means that 806 patches are duplicates. After in-depth analysis, we find the reasons for such a phenomenon are twofold. On one hand, current APR tools tend to repeatedly generate correct patches for a small number of bugs, as revealed by Liu et al. [51]. Specifically, we find that one correct patch can be generated by nine APR tools and seven correct patches can be generated by at least five tools (there are in total 11 repair tools in the RepairThemAll framework). On the other hand, the previous study [76] mistakenly repeats the correct patch of Defects4J bug Closure-126 generated by Arja [97] for more than 700 times in the dataset.[1] This highlights that large scale empirical evaluation needs to be carefully checked. Consequently, we make all the source code and data in this study publicly available in our online repository[2] to facilitate the review from the community.

Statistics on the two datasets we constructed are listed in Table 1. It should be noted that they are both balanced.

4.2.2 [RQ-1]. Note that in the previous study [76], authors use pre-trained models BERT, Doc2Vec, and CC2Vec to represent buggy and patched lines, and train three classification models (i.e., Decision Tree[8], Logistic Regression[32], and Naive Bayes[66]) using their data. Following such studies, we reuse their pre-trained representation learning models and train the classifiers based on our data. We also include two other representation techniques code2vec [5] and code2seq [2] in this study that are excluded by the previous study [76]. Note that although they take the program structure into consideration, they are not designed for embedding code changes and patches. We use them to encode the buggy and patched methods for representing buggy and patched code and then also train the classifier based on the obtained vectors.

For all the methods used for comparison, we perform three experiments: first, we conduct 5-fold cross-validation on the small dataset following the previous study; second, we also conduct 5-fold cross-validation on the large dataset; at last, we retrain these models on the large dataset and test them on the small dataset (note that we exclude duplicated patches from the large dataset when

[1]This can be checked through their online repository at: https://github.com/TruX-DTF/DL4PatchCorrectness/tree/master/data/experiment2.
[2]https://github.com/Ringbo/Cache.

performing this experiment to avoid data leakage). The first two experiments separately assess the performances on the small and large datasets, while the third one assesses the generality of each learned model.

4.2.3 [RQ-2]. Wang et al. [81] performed a systematic study and built baselines for existing APCA techniques (except ODS) on their 902 patches. We note that running dynamic APCA techniques is quite time-consuming. We thus in this RQ directly reuse the patch benchmark from Wang et al. and also the associated evaluation results for all the existing APCA techniques (i.e., we do not re-execute dynamic techniques such as Daikon and Evosuite). Wang et al. [81] also selected eight static patch features w.r.t code token sequence, code syntax, as well as code semantics (two from ssFix [86], three from S3 [37], and three from CapGen [82]) for differentiating overfitting patches from correct ones. The authors investigated the effectiveness of utilizing machine learning models to assess patch correctness based on static code feature values under the setting of 10-fold cross-validation where *Random Forest* achieved the best performance. Also, note that the latest APCA technique, ODS [92], was also evaluated on this benchmark using a 10-fold cross validation. Therefore, to fairly compare with all the state-of-the-art, we also perform 10-fold cross-validation for Cache on these patches. Note that during 10-fold cross-validation, the final performance of our model is summed up over the 10 rounds of each training and testing process and the same to the above-mentioned 5-fold cross-validation process.

4.2.4 [RQ-3]. We split factors that contribute to our approach from two aspects: those from our embedding model and those from our integration functions.

For the embedding model, we dissect the impacts from (1) splitting code tokens into subtokens, (2) LSTM-based AST path encoder, (3) the attention architecture, (4) the context within the buggy method, (5) the program structure preserved by the AST nodes, and (6) separately embedding the changed part of the code and the surrounding context. To assess the impact of sub-token splitting, we directly feed each token to $E_t(*)$. To assess the impact of AST path encoder, we learn an embedding matrix E_{path} to represent the whole path instead of embedding each node in the path and feeding them into LSTM. To assess the impact of the attention network, we directly drop out the attention network and use the outputs from the fully-connected layer as the representations of an AST. To assess the impact of contexts, we design another pipeline in which the context (i.e., E_{Con}) is removed and thus E_{B_m} and E_{P_m} simply equal to E_{B_t} and E_{P_t}, respectively. To assess the impact of program structures embedded in the AST node, we directly remove the LSTM-based path encoder and use two tokens from the leaf nodes for representing an AST path. To assess the impact of splitting the changed code from its context, we use the AST encoder to directly encode the buggy and patched methods and feed them into the patch classification step.

We employ five functions in total to integrate the representations of the buggy method and patched method for patch classification. To assess the contribution of each of them, we remove one of them and evaluate the effectiveness when the other four functions are kept in our model.

4.3 Model Tuning

Recall that our large dataset contains a total of nearly 50,000 patches. Therefore, it will be time-consuming if we use all of them for model tuning. We thus randomly selected 10k patches to fine-tune our model. We mainly focused on the batch size (64, 128, 256), learning rate (0.01, 0.02, 0.05), and dropout (0.1, 0.2, 0.5). We experimented with all combinations of the listed parameter values. Finally, the batch size, learning rate, and dropout are empirically determined to 128, 0.01, 0.2, respectively. Note that for the AST encoder part, the hyper-parameters such as the maximum number of nodes in an AST path and the dimension of the output vector are reused from code2seq [2]. Also, hyper-parameters for baseline techniques are already fine-tuned by Tian et al. [76].

Table 2. Effectiveness of Cache Compared with Other Representation Learning Techniques on the Small Dataset

Classifier	Embedding	Acc.	Pre.	Recall.	F1	AUC
Decision Tree [8]	BERT [15]	63.5	65.3	70.9	67.9	63.7
	CC2vec [26]	66.1	69.4	68.0	68.7	66.5
	code2vec [5]	65.1	68.1	68.3	68.1	64.4
	code2seq [2]	60.1	63.5	64.0	63.7	60.0
	Doc2Vec [35]	61.2	64.5	65.3	64.8	60.8
Logistic Regression [32]	BERT [15]	64.8	66.5	72.4	69.2	68.7
	CC2vec [26]	64.9	62.4	90.1	73.7	68.6
	code2vec [5]	66.8	68.6	72.9	70.6	70.2
	code2seq [2]	60.7	63.3	67.6	65.3	63.1
	Doc2Vec [35]	63.7	65.7	70.8	68.0	68.9
Naïve Bayes [66]	BERT [15]	61.6	64.8	65.7	65.0	64.7
	CC2vec [26]	60.0	58.3	**94.6**	72.2	58.1
	code2vec [5]	57.7	58.1	81.5	67.8	55.6
	code2seq [2]	57.0	59.0	70.5	64.2	60.6
	Doc2Vec [35]	64.1	65.8	72.4	68.7	67.0
Cache		**75.4**	**79.5**	76.5	**78.0**	**80.3**

4.4 Evaluation Metrics

For evaluating the performances of various approaches, we adopt classical metrics used for assessing classification models: Accuracy, Precision, Recall, and F1-score.

Suppose we have $P_{correct}$ correct patches and $P_{overfitting}$ overfitting patches in the dataset. We measure the accuracy, precision, recall, and F1-score to answer the designed RQs, whose definitions are provided by the previous study [81]: **True Positive (TP):** An overfitting patch in $P_{overfitting}$ is identified as overfitting. **False Positive (FP):** A correct patch in $P_{correct}$ is identified as overfitting. **False Negative (FN):** An overfitting patch in $P_{overfitting}$ is identified as correct. **True Negative (TN):** A correct patch in $P_{correct}$ is identified as correct.

$$Accuracy \leftarrow (TP + TN)/(TP + FP + FN + TN)$$
$$Precision \leftarrow TP/(TP + FP)$$
$$Recall \leftarrow TP/(TP + FN)$$
$$F1 - score \leftarrow \frac{2 * Precision * Recall}{Precision + Recall}$$

For representation learning techniques, we also assess their **AUC (Area Under Curve)** values, which is also a widely used metric to evaluate classification algorithms [6, 22, 76].

5 STUDY RESULTS

5.1 RQ1: Comparing with Representation Learning Techniques

Comparison results against previous representation learning techniques are presented in Table 2 to Table 4. We find that Cache performs the best under each experimental setting.

On the small dataset, Cache is around 6.0% and 14.4% higher than the state-of-the-art with respect to the F1-score (78.0% vs. 73.7% from CC2vec plus Logistic Regression) and AUC (80.3% vs. 70.2% from code2vec plus Logistic Regression). As illustrated, the accuracy and precision values of Cache are always the leading one compared against other strategies. Particularly, the values of Cache on these two metrics are 75.4% and 79.5%, respectively, while the optimum values from all

Table 3. Effectiveness of CACHE Compared with Other Representation Learning Techniques on the Large Dataset

Classifier	Embedding	Acc.	Pre.	Recall.	F1	AUC
Decision Tree [8]	BERT [15]	95.7	93.9	97.4	95.6	95.9
	CC2vec [26]	95.6	95.4	95.7	95.5	95.7
	code2vec [5]	95.0	93.2	96.6	94.9	95.4
	code2seq [2]	92.2	91.0	93.2	92.3	92.4
	Doc2Vec [35]	85.1	84.2	85.3	84.7	85.3
Logistic Regression [32]	BERT [15]	82.4	83.6	79.4	81.4	91.0
	CC2vec [26]	91.2	96.1	85.4	90.4	95.0
	code2vec [5]	89.6	88.6	90.2	89.4	95.0
	code2seq [2]	91.5	90.5	92.2	91.4	96.0
	Doc2Vec [35]	90.4	91.9	88.0	89.9	96.1
Naïve Bayes [66]	BERT [15]	68.2	80.3	45.7	58.2	74.6
	CC2vec [26]	78.4	94.8	58.6	72.5	92.4
	code2vec [5]	61.4	68.7	37.4	48.4	69.3
	code2seq [2]	70.3	76.8	55.5	64.5	78.9
	Doc2Vec [35]	81.2	86.4	75.5	78.9	88.9
CACHE		**98.6**	**98.9**	**98.2**	**98.6**	**98.9**

other approaches are 66.8% and 69.4%. This suggests that CACHE can generally achieve the most accurate predictions and the patches identified as overfitting by CACHE are of high confidences to be overfitting. As for recall, values from CC2vec and code2vec can sometimes exceed that of CACHE since they tend to classify most patches as overfitting (e.g., CC2vec with Naïve Bayes classifies 1,051 out of 1,183 patches as overfitting and thus achieves a high recall of 94.6%). However, these approaches achieve relatively low precision. In contrast, CACHE can achieve a high recall exceeding 75% while preserving a high precision of 79.5%.

On the large dataset, CACHE still outperforms other approaches on each metric. For instance, for the F1-score, CACHE reaches 98.6%, which is more than 3% higher than the second highest value (i.e., 95.6%) obtained from BERT with the Decision Tree. A notable phenomenon here is that for each method, the performance achieved over the large dataset is generally higher than that achieved on the small dataset. CACHE is a vivid example whose F1 score skyrockets from 78% to 98.6%. We attribute the performance improvement to the fact that the large dataset is structurally simpler. Specifically, all patches from the ManySStuBs4J dataset are single-line patches while patches in the small dataset can sometimes cross multiple lines (e.g., only 52.4% developer-provided patches from Defects4J are single-line patches [72]). Such simplicity can make it easy for the network to capture hidden features.

When training on the large dataset while testing on the small dataset (cf. Table 4), we witness performance decreases for each approach compared against the results achieved over 5-fold cross-validation on either the small dataset or the large dataset. A reasonable explanation here is that training on a dataset where patches are structurally simple (as we have mentioned above) reduces the generalization of learned models. Nevertheless, CACHE still achieves the optimum overall performance with the F1-score reaching 75% and AUC reaching 73.2%, much higher than those of the baselines. What needs attention is that CC2Vec achieves rather low recalls with various classifiers. For instance, when integrated with Logistic Regression, the number of TP is only 2 while the number of FN is 646, leading to a low recall of 0.3%. In an extreme condition, it identifies all patches as correct and thus its recall is 0% (when integrated with Naïve Bayes). We attribute this phenomenon to the low generalization ability of representing patches as token sequences. CACHE

Table 4. Effectiveness of Cache Compared with Other Representation Learning Techniques When Training and Testing Across Datasets

Classifier	Embedding	Acc.	Pre.	Recall.	F1	AUC
Decision Tree [8]	BERT [15]	59.7	65.9	54.6	59.7	59.9
	CC2vec [26]	46.2	52.1	22.7	31.6	47.7
	code2vec [5]	59.5	64.0	60.6	62.3	62.9
	code2seq [2]	59.9	63.3	63.9	63.6	59.6
	Doc2Vec [35]	49.3	54.4	46.5	50.1	47.8
Logistic Regression [32]	BERT [15]	53.6	56.4	67.5	61.5	55.9
	CC2vec [26]	44.9	9.0	0.3	0.6	41.9
	code2vec [5]	57.8	61.5	61.3	61.4	59.1
	code2seq [2]	57.9	59.8	70.5	64.7	57.1
	Doc2Vec [35]	47.0	52.0	41.4	46.1	47.3
Naïve Bayes [66]	BERT [15]	46.7	52.1	34.8	41.8	49.4
	CC2vec [26]	45.2	NaN	0.0	NaN	50.0
	code2vec [5]	42.4	43.9	18.5	26.1	46.4
	code2seq [2]	48.1	54.1	34.6	42.2	50.7
	Doc2Vec [35]	41.8	44.8	27.5	34.1	43.6
Cache		**70.1**	**69.1**	**81.9**	**75.0**	**73.2**

is much more generalizable since the precision and recall values are balanced (69.1% and 81.9%, respectively). Such results reveal that Cache can identify a large amount of the overfitting patches while preserving a high confidence that a patch identified as overfitting is indeed overfitting. It outperforms the state-of-the-art by around 16% with respect to the F1-score (75% vs. 64.7%). Such results further reflects the fact that considering the context information and program structures is important to assess patch correctness.

> *Cache can outperform other representation learning based techniques significantly under different experimental settings. Specifically, it achieves an average F1-score of 78.0% on the small dataset and 98.6% on the large dataset, which outperforms existing techniques by at least 6% and 3%, respectively. More importantly, it achieves better generalities across different datasets, which surpass existing techniques by around 16%.*

5.2 RQ2: Comparing with APCA Techniques

Recently, Wang et al. [81] systematically evaluated and compared the effectiveness of existing APCA techniques. They split the state-of-the-art from two aspects: whether they need to execute tests (dynamic vs. static) and whether they need the ground-truth patch. Cache does not need the oracle nor running test cases, thus belonging to the *static* and *no oracle required* types. The experiment results are listed in Table 5.

We note that Cache achieves the best overall performance with the F1 score reaching 93.7%, the only one that exceeds 90%, while the previous leading one (i.e., the Random Forest strategy) reaches 88%. For other metrics, Cache also possesses the highest accuracy and recall values. Specifically, its accuracy and recall values are the only ones that exceed 90%. As for precision, Cache not only beats other static approaches but also performs better than three dynamic techniques, which are *Daikon*, *PATCH-SIM*, and *E-PATCH-SIM*. We consider this result as acceptable since executing tests may help dynamic techniques enhance their precision [81, 92], while Cache does not need any

Table 5. Effectiveness of Each APCA Technique on Dataset from Wang et al. [81]

APCA	Acc.	Prec.	Recall.	F1
Evosuite [23]	65.9	99.1	53.5	69.5
Randoop [65]	51.3	97.4	33.8	50.2
DiffTGen [85]	49.6	97.4	30.6	46.6
Daikon [20]	76.1	89.9	73.7	81.0
R-Opad [91]	34.9	**100.0**	10.2	18.5
E-Opad [91]	37.7	**100.0**	14.7	25.6
PATCH-SIM [87]	49.5	83.0	38.9	53.0
E-PATCH-SIM [81]	41.7	82.1	25.8	39.3
Anti-patterns [74]	47.6	85.5	33.5	48.1
S3 [37]	69.7	79.3	78.9	79.0
ssFix [86]	69.2	78.9	78.8	78.8
CapGen [82]	68.0	78.3	77.4	77.8
Random Forest [25]	72.5	87.0	89.1	88.0
ODS [92]	88.9	90.4	**94.8**	92.5
Cache	**90.8**	92.9	94.5	**93.7**

▨ denotes techniques that require the oracle information. The bold name means the technique is dynamic.

run-time information. We also note that although *Daikon* requires the oracle information, its precision is still over-performed by Cache, which reveals the effectiveness of our approach.

We also compare Cache with ODS [92] and the results are illustrated in the last two lines of Table 5. We note that although the recall of ODS slightly exceeds that of Cache (94.8% vs. 94.5%), Cache still outperforms ODS with respect to Accuracy (90.8% vs. 88.9%) and F1-score (93.7% vs. 92.5%). Such results are even more promising considering that ODS needs to mine hundreds of manually-designed patch features that explicitly target patch correctness assessment while the patch semantics are automatically learned by Cache. The captured semantics of Cache can also be applied to other downstream tasks related with code changes.

As a static technique, Cache achieves the optimum overall performance compared with existing APCA techniques with a high F1-score reaching 93.7%. Furthermore, it can even achieve a higher precision than certain dynamic techniques, e.g., PATCH-SIM.

5.3 RQ3: Contribution of Each Factor

Results of the ablation study are demonstrated in Table 6. Generally speaking, all our model designs make contributions to the final performance, more or less. For example, if we do not hold the program structure information embedded in AST nodes on the small dataset, the overall performance with respect to the F1-score will be decreased by 8.5%.

We note that the context information contributes the most to the overall performance of Cache without which the F1-score will degrade the most for all three experiments. For instance, if the context information is not included, the F1-score of Cache will be decreased by 12.4% on the small dataset. This demonstrates the rationale of our motivation that is the context information can help better assess the correctness of a code change (i.e., a patch). We also note if we do not split code tokens into sub-tokens or do not use an LSTM encoder for embedding AST paths, the performances will also drop at notable degrees (e.g., 9.2% and 10.1%, respectively on the small dataset). This indicates that the information hidden in the AST should be carefully extracted and learned. For heuristics we use for integrating representations, the cosine similarity appears to be

Table 6. Performance of Variants of Cache Under Diverse Experiment Settings

Dataset / Model	Small		Large		Cross-dataset	
	F1	↓ (%)	F1	↓ (%)	F1	↓ (%)
No Context	68.3	**12.4**	96.5	**2.1**	67.9	**9.5**
No token splitting	70.8	9.2	97.4	1.2	69.4	7.5
No LSTM encoder	70.1	10.1	98.1	0.5	69.2	7.7
No attention	70.5	9.6	97.2	1.4	68.5	8.7
No program structure	71.4	8.5	98.0	0.6	68.8	8.3
No split-embedding	72.1	8.2	98.2	0.4	70.1	6.5
No addition	71.9	7.8	98.2	0.4	70.2	6.4
No subtraction	76.4	2.1	98.5	0.1	71.8	4.3
No Hadamard product	72.3	7.3	98.3	0.3	70.1	6.5
No cosine similarity	70.8	9.2	97.9	0.7	69.2	7.7
No Bilinear model	75.2	3.6	98.4	0.2	71.5	4.7
Cache (original model)	78.0		98.6		75.0	

↓ denotes performance degradation.

the most rewarding one without which the F1-score decreases by 9.2%, 0.7%, 7.7% under three experiment settings, respectively.

All the design decisions in Cache contributed to its outstanding performance (e.g., outperforms the state-of-the-art by around 16% in the cross-dataset evaluation), among which the context information is the most rewarding one. Specifically, if ignoring the context information, Cache will suffer from decreases of 12.4%, 2.1%, and 9.5% w.r.t the F1-score under three evaluation settings, respectively.

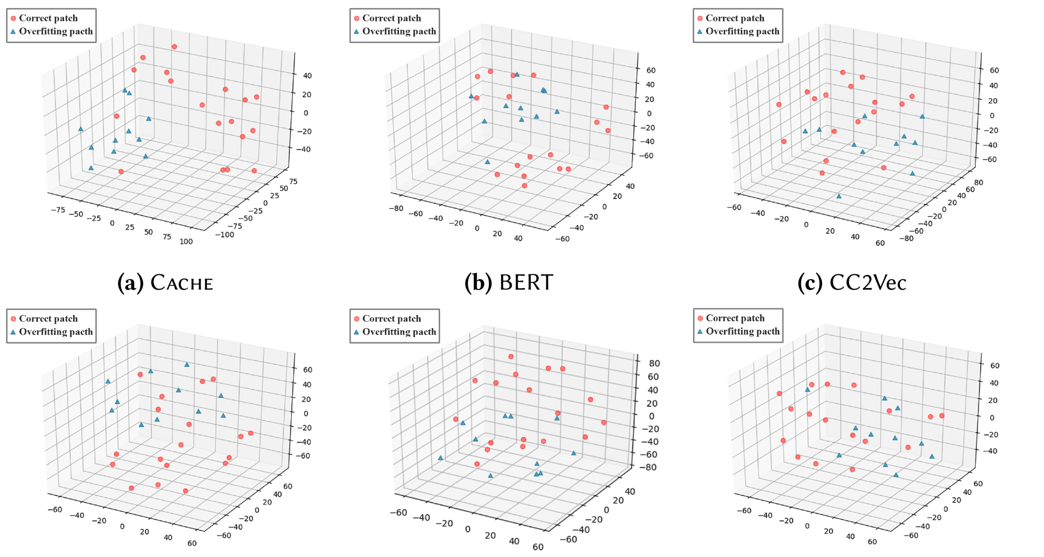
6 DISCUSSION

6.1 Contribution of the Embedding Model

We note that compared with the previous study [76], which uses four features and machine learning based classifiers to determine the patch correctness, our patch classification differs in two aspects: we extract five features and adopt a deep learning based classifier. However, the superiority of Cache is still majorly attributed by the embedding mechanism. We demonstrate this via the following experiments.

We pre-trained Cache on the large dataset. We then performed three experiments on the small dataset (also with 5-fold cross-validation) where (1) we replaced our five features with the four features in [76], (2) we replaced our deep learning based classifier with a Logistic Regression classifier, and (3) we replaced both the features and the classifier. Results reveal that we can obtain F1-scores of 75.2%, 75.0%, and 74.3%, respectively, under three conditions. Such performances are all higher than those from all the state-of-the-art (70.6% if ignoring the figure from CC2vec which tends to identify every patch as overfitting) but only slightly lower than that from Cache (78.0%). Such results indicate that our context-aware embedding approach can better capture the code change information than the state-of-the-art and contribute significantly to the performance of Cache. Our specially designed classification stage helps Cache reach the optimum.

During the first experiment (i.e., features used to integrate the vectors are identical for all the embedding techniques), we also visualized the vectors embedded by different techniques before feeding them into the final classifiers. To perform a pairwise comparison (i.e., classification results of overfitting and correct patches based on the same piece of buggy code), we chose patches for



(a) Cache **(b)** BERT **(c)** CC2Vec

(d) code2seq **(e)** code2vec **(f)** Doc2vec

Fig. 5. Distribution of embedding vectors generated by Cache, BERT, CC2Vec, code2seq, code2vec and Doc2vec for patches of Math-5.

Math-5 as the study objects since the numbers of correct and overfitting patches for this bug are considerable (11 and 18, respectively). We extracted feature vectors generated by Cache, BERT, CC2Vec, code2seq, code2vec and Doc2Vec, respectively. The dimensions of the vectors produced by these six techniques are 642 ($2n+2$, n = 320), 2050 ($2n+2$, n = 1024), 130 ($2n+2$, n = 64), 642 ($2n+2$, n = 320), 642 ($2n+2$, n = 320), 130 ($2n+2$, n = 64), respectively. We projected these high-dimensional vectors into a three-dimensional space with the help of the t-SNE technique [58]. Results are shown in Figure 5. From the figure, we observe that the overfitting patches can be better identified by Cache: feature vectors of the correct patches generated by Cache are majorly gathered in the upper right part while those of the overfitting patches are in the bottom left part. As for comparison, such a clear segmentation has not been witnessed for the results generated by other tools: we observe a mixture of correct and overfitting patches in the other five sub-figures. Such results again demonstrate that the context-aware embedding, which leads to easily-identified feature vectors, is the main contribution for the performance of Cache. The achieved performances of Cache reveal that considering the context information for representing code changes is reasonable. This suggests that future works towards this direction could focus more on how to utilize the context information to better understand and characterize the code changes.

6.2 The Capability of Identifying Overfitting Patches When No Correct Patch is Misclassified

It should be noted that in patch classification, precision is considered as more important than recall (i.e., we should avoid misclassifying correct patches) [81, 92, 96]. Specifically, Xiong et al. [87] and Tian et al. [76] respectively designed an experimental setting to avoid misclassifying any correct patches when evaluating PATCH-SIM and existing representation learning techniques. In our study, we also performed a similar experiment where we trained Cache on the 870 patches used by Tian et al. and evaluated it on the 139 patches used by both Xiong et al. and Tian et al.. Such

Table 7. Comparison of Overfitting Patch Identification Between PATCH-SIM, BERT+LR, and Cache

	Ground Truth		PATCH-SIM [87]		BERT+LR [76]		Cache	
Project	**Overfitting**	**correct**	**Overfitting excluded(%)**	**correct excluded**	**Overfitting excluded(%)**	**correct excluded**	**Overfitting excluded(%)**	**correct excluded**
Chart	23	3	14(60.9%)	0	16(69.6%)	0	18(78.3%)	0
Lang	10	5	6(54.5%)	0	1(10%)	0	3(30%)	0
Math	63	20	33(52.4)	0	23(36.5%)	0	32(50.8%)	0
Time	13	2	9(69.2%)	0	3(23.1%)	0	2(15.4%)	0
Total	109	30	62(56.3%)	0	43(39.4%)	0	55(50.4%)	0

a decision ensures a fair comparison against existing representation learning techniques studied by Tian et al.. Results are shown in Table 7. In this table, the Logistic Regression with BERT embedding is shown since it achieves the state-of-the-art performance among existing representation learning techniques. The threshold of Cache was set to 0.858 (i.e., a patch is considered as overfitting if $p(\mathbf{E}_{\text{out}_i})$ is larger than 0.858) since Cache can avoid misclassifying any correct patch under such a setting. We find that Cache is able to exclude more than half of the overfitting patches in this dataset (i.e., 55). As for comparison, the state-of-the-art representation learning technique only excludes 43 overfitting patches and PATCH-SIM excludes 62 ones, which is only slightly higher than Cache. We also note for patches from a specific project (i.e., Chart), Cache can exclude more overfitting patches than PATCH-SIM. This means a reasonable achievement for Cache since in contrast to PATCH-SIM, it does not need to execute any test.

6.3 The Rationale of 5-Fold Cross-Validation

We use cross-validation in our evaluation by following previous studies [76, 81]. However, as suggested by [57], cross-validation may be biased for time-sensitive data like patches in Defects4J. Therefore, we performed another experiment on the small dataset where we used patches submitted before September, 2012 for training and the rest for testing. Such a separation leads to 954 training data and 229 testing data. We obtained an F1-score of 78.1%, nearly identical to what we obtained through 5-fold cross-validation (78.0%, cf. Table 2). As a result, impacts from the time-sensitive data for our cross-validation are rather limited.

We also calculated the standard deviation of the per-fold F1 values for each technique during the 5-fold cross-validation on small and large datasets. Results are shown in Table 8. We note that the biggest deviation values on the two datasets are both from BERT with Naïve Bayes (2.87 and 0.91, respectively) and for other techniques, the deviation values are rather small. For instance, the deviation values of Cache are 1.16 and 0.23, respectively. Such results suggest that the achieved F1-scores of our included techniques on each fold are similar and the performance deviation can be rather limited.

6.4 Threats to Validity

6.4.1 External Validity. One threat is the representativeness of our considered representation learning techniques among the diverse models. This is mitigated by the fact that they include both code-oriented models (e.g., CC2vec) and natural language specific ones (e.g., BERT). Besides, we include more representation learning techniques (e.g., code2seq) than the previous study [76] to mitigate such threats.

6.4.2 Internal Validity. In this study, we compared Cache with a wide range of existing techniques including dynamic and static APCA techniques as well as representation learning techniques. The implementation of the baselines posts threats to the internal validity. To alleviate this

Table 8. The Standard Deviation of the Per-fold F1-values for Each Technique During the 5-fold Cross-validation on Small and Large Datasets

Classifier	**Embedding**	**Small**	**Large**
Decision Tree [8]	BERT [15]	0.90	0.15
	CC2vec [26]	1.92	0.47
	code2vec [5]	1.53	0.19
	code2seq [2]	0.67	0.35
	Doc2Vec [35]	1.79	0.34
Logistic Regression [32]	BERT [15]	1.58	0.47
	CC2vec [26]	0.84	0.12
	code2vec [5]	1.74	0.25
	code2seq [2]	1.29	0.19
	Doc2Vec [35]	1.17	0.08
Naïve Bayes [66]	BERT [15]	2.87	0.91
	CC2vec [26]	1.59	0.90
	code2vec [5]	1.62	0.90
	code2seq [2]	1.84	0.81
	Doc2Vec [35]	2.17	0.51
Cache		1.16	0.23

threat, we chose to reuse the fine-tuned representation learning models from Tian et al. [76] and reuse the reported performances of existing APCA techniques from Wang et al. [81].

Regarding the large dataset we collected, the correct patches are largely from ManySStuBs4J while the overfitting patches are from RepairThemAll. There is a chance that models learned from this dataset simply distinguish ManySStuBs4J from RepairThemAll. However, Cache is able to distinguish correct patches from overfitting ones instead of distinguishing from different datasets. Specifically, We demonstrate this via two aspects. First, in our cross-dataset evaluation where we trained models on the large dataset and tested them on the small one, Cache achieved an F1-score of 75% and significantly outperformed other techniques. This experiment can exactly address this concern in that patches in our small dataset are all from Defects4J and Defect4J is a part of RepairThemAll. If Cache simply learns to distinguish the two benchmarks used in the experiment, it will tend to identify patches from Defects4J as overfitting, but that is not the fact. Second, we also checked the experiment results of the 5-fold cross-validation on the large dataset. We found that for the 94 correct patches from the RepairThemAll (after deduplication), Cache identifies 71% of them as correct while BERT with Decision Tree (which achieves the best performance among the state-of-the-art in this experiment, cf. Table 3) only identifies 17% of them as correct. These results further illustrate that (1) the state-of-the-art representation learning techniques tend to distinguish patches according to the datasets, but (2) the effectiveness of Cache is not influenced by the dataset difference significantly.

Another threat comes from the patches selected in our datasets. Specifically, patches from the ManySStuBs4J dataset are mined from bug-fixing commits of GitHub projects' historical information. They may include unrelated changes such as refactoring. However, this threat is mitigated considering that all the code changes in the commit should be able to match a fixing template defined in previous APR studies [40, 54], otherwise the commit will be omitted. Also, patches in our

Table 9. The Performance of Cache on Patches from Each Individual APR Tool

Tool	Acc.	Prec.	Recall.	F1
ACS [88]	80.5	58.3	70.0	63.6
Arja [97]	73.7	94.7	73.5	82.8
AVATAR [49]	81.5	93.5	78.4	85.3
CapGen [82]	92.0	97.4	92.7	95.0
Cardumen [62]	77.8	100.0	77.8	87.5
DynaMoth [17]	81.8	100.0	80.9	89.5
FixMiner [34]	84.0	100.0	78.9	88.2
GenProg-A [97]	88.0	100.0	87.5	93.3
HDRepair [38]	75.0	75.0	75.0	75.0
Jaid [12]	73.6	75.6	77.5	76.5
jGenProg [61]	89.7	96.8	90.9	93.8
jKali [61]	82.9	90.3	90.3	90.3
jMutRepair [61]	68.7	84.6	78.5	81.5
Kali-A [97]	65.8	96.0	66.7	78.7
kPAR [48]	64.7	100.0	62.5	76.9
Nopol [89]	86.7	100.0	86.2	92.6
RSRepair-A [97]	78.8	96.1	80.6	87.7
SequenceR [13]	74.5	86.0	82.2	84.1
SimFix [28]	58.6	80.0	57.1	66.7
SketchFix [27]	100.0	100.0	100.0	100.0
SOFix [52]	63.6	20.0	100.0	33.3
TBar [50]	85.0	93.5	87.9	90.6

datasets may be mislabelled since they are manually assessed by previous studies [76, 81]. This is mitigated considering that all patches are labeled following the same criteria provided by [51] and all data is publicly available to the community for review.

There is also a concern that if a large number of patches within our small dataset are generated by a specific APR tool, our Cache may work well on such patches while perform poorly on other patches since patches generated by different tools may exhibit different patterns. This is mitigated through two aspects of analysis. First, as revealed by the Table 3 of the ODS study [92], the number of patches generated by different APR tools are generally balanced. Second, we also investigated the performance of Cache on patches from each individual APR tool and demonstrated the results in Table 9. In general, Cache achieves similar performances on patches from different APR tools with F1-scores ranging from 70% to 90% under most conditions. We note that Cache achieves an F1-score of 33.3% on patches from SOFix. This is potentially because the number of patches from the tool is relatively small (i.e., a total of 11).

6.4.3 Reliability. We have built an online repository for Cache at https://github.com/Ringbo/Cache, where all the data and source code of our study are publicly available. Moreover, we have provided step-to-step guidance for reproducing our results.

7 RELATED WORK

Deep Learning in APR. Deep learning techniques have been widely used in recent APR researches. For fault localization, Zhang et al. [100] proposed to use **convolutional neural**

networks (CNN) to learn suspiciousness of program elements in traditional spectrum-based fault localization [1]. Zou et al. [102] propose *CombineFL* which applies learning-to-rank model [42] to combine different types of fault localization techniques with the aim to achieve the optimum performance. For patch generation, Chen et al. [13] introduced SequenceR, which adopts a seq2seq model to learn the code transformation. It considers the patching task as a translation problem and generates an entirely new statement from a buggy one. When it comes to patch validation, Csuvik et al. [14] were the first to propose to utilize embedding approaches for assessing patch correctness, which was further evaluated by Tian et al. [76]. However, their works only focus on previous embedding models. Our study, as comparison, is the first to propose a specialized embedding model for patch correctness assessment that incorporates the context information and program structures, both of which are missed by the previous two studies. Moreover, we build two patch datasets in our study and assess the generality of the learned models via cross-dataset evaluation while the two previous studies focus on small-scale patch datasets only.

AST Path for Software Engineering. Alon et al. [4] first pointed out that using paths in the program's AST significantly lowers the learning effort (compared to learning over program text) and is still scalable and general. After that, this code representation technique has been exploited by numbers of state-of-the-art approaches in **software engineering (SE)** such as code2vec [5] which predicts names for methods, code2seq [2] which focuses on code summarization and code captioning, AnyCodeGen [3] which can generate a missing piece of source code without any restriction on the vocabulary or structure, and so on. Our study adopts these code representation techniques due to its demonstrated effectiveness in various software engineering tasks.

Context in Program Repair. From results in other works reported by repair community, taking context into consideration can always help make the approaches more solid. In the early stage of program repair, fixing ingredients (i.e., program elements utilized for generating the patch) were randomly selected [40, 67], leading to a low precision (i.e., patches pass the tests are still highly likely to be incorrect). Wen et al. [82] took into consideration the buggy code context and calculated similarity for selecting fixing ingredients. By doing so, fixing ingredients that are more likely to be used for generating correct patches can be prioritized. Therefore, their APR tool, *CapGen*, is the most precise one so far. *CoCoNut* [57], the recently-proposed deep learning-based APR technique, separately encodes buggy lines and context surrounding the fixing locations. It thus achieves better performance than precious works which ignore context information [13]. *Getafix* [7] summarizes past fix patterns and uses the context of candidate patches to select the most appropriate fix, avoiding the exploration of a large space. Our work utilizes context information for code change embedding and also gets the optimum performance so far on patch correctness assessment.

Context in Other SE Tasks. The context information also plays critical role for solving SE tasks beyond program repair. Brody et al. [9] utilize the code edits that are already done within a file as *context* information to predict edits to be performed in the rest of the file. Li et al. [43] consider the caller/callee relation as *context* and enhance the effectiveness of bug prediction. Wang et al. [80] utilize the caller/callee relation in method name recommendation and also obtain the state-of-the-art effectiveness. Wu et al. [84] focus on the data flow information to precisely detect *reentrancy* vulnerabilities among smart contracts. Given a source AST which serves as the *context*, Chakraborty et al. [10] use a tree-to-tree neural translation model to predict the code change pattern. We are the first to utilize the unchanged code within a method as the *context* information for embedding code change.

8 CONCLUSION

We present a new code change embedding approach where (1) we take the unchanged code into consideration to leverage context information, and (2) we adopt the *AST path* technique for embedding to capture the structure information from AST nodes. We implement the idea as Cache and apply it on the patch correctness assessment task. We explore the performance of Cache using various experimental settings and demonstrate that Cache can outperform both existing representation learning techniques and the state-of-the-art APCA approaches. Our future work would focus on exploring the effectiveness of context-aware code change embedding on other tasks such as log message generation and just-in-time defect prediction.

Artefacts: All data and code in the study are publicly available at:

https://github.com/Ringbo/Cache.

ACKNOWLEDGMENTS

The authors wish to thank the anonymous reviewers for their constructive comments.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*. IEEE, 89–98. https://doi.org/10.1109/TAIC.PART.2007.13

[2] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating sequences from structured representations of code. In *Proceedings of the 7th International Conference on Learning Representations*. OpenReview.net.

[3] Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2020. Structural language models of code. In *Proceedings of 37th International Conference on Machine Learning*.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. A general path-based representation for predicting program properties. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 404–419. https://doi.org/10.1145/3192366.3192412

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40:1–40:29. https://doi.org/10.1145/3290353

[6] Abdulrahman Alshammari, Christopher Morris, Michael Hilton, and Jonathan Bell. 2021. FlakeFlagger: Predicting flakiness without rerunning tests. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1572–1584. https://doi.org/10.1109/ICSE43902.2021.00140

[7] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 159:1–159:27. https://doi.org/10.1145/3360585

[8] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. 1984. *Classification and Regression Trees*. Routledge. https://doi.org/10.1201/9781315139470

[9] Shaked Brody, Uri Alon, and Eran Yahav. 2020. A structural model for contextual code changes. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[10] Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. 2020. CODIT: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering* (2020).

[11] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and precise on-the-fly patch validation for all. In *Proceedings of the 43rd International Conference on Software Engineering*.

[12] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 637–647. https://doi.org/10.1109/ASE.2017.8115674

[13] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Trans. on Software Engineering* (2019).

[14] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. 2020. Utilizing source code embeddings to identify correct patches. In *Proceedings of the 2nd International Workshop on Intelligent Bug Fixing*. IEEE, 18–25. https://doi.org/10.1109/IBF50092.2020.9034714

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 4171–4186. https://doi.org/10.18653/v1/n19-1423

[16] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 302–313. https://doi.org/10.1145/3338906.3338911

[17] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop in Automation of Software Test*. ACM, 85–91. https://doi.org/10.1145/2896921.2896931

[18] Thomas Durieux and Martin Monperrus. 2016. IntroClassJava: A benchmark of 297 small and buggy Java programs. In *Technical Report #hal-01272126*. University of Lille.

[19] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123.

[20] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering* 27, 2 (Feb. 2001), 99–123.

[21] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. ACM, 313–324. https://doi.org/10.1145/2642937.2642982

[22] Yuanrui Fan, Xin Xia, David Lo, and Ahmed E. Hassan. 2018. Chaff from the wheat: Characterizing and determining valid bug reports. *IEEE Transactions on Software Engineering* 46, 5 (2018), 495–525.

[23] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 416–419.

[24] Anbang Guo, Xiaoguang Mao, Deheng Yang, and Shangwen Wang. 2018. An empirical study on the effect of dynamic slicing on automated program repair efficiency. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 554–558.

[25] Tin Kam Ho. 1995. Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, Vol. 1. IEEE, 278–282.

[26] Thong Hoang, Hong Jin Kang, Julia Lawall, and David Lo. 2020. CC2Vec: Distributed representations of code changes. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 518–529. https://doi.org/10.1145/3377811.3380361

[27] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 12–23. https://doi.org/10.1145/3180155.3180245

[28] 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*. ACM, 298–309. https://doi.org/10.1145/3213846.3213871

[29] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 23rd International Symposium on Software Testing and Analysis*. ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[30] Rafael-Michael Karampatsis and Charles A. Sutton. 2020. How often do single-statement bugs occur? the ManySStuBs4J dataset. In *Proceedings of the 17th Mining Software Repositories*. IEEE. http://arxiv.org/abs/1905.13334.

[31] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[32] David G. Kleinbaum, K. Dietz, M. Gail, and Mitchel Klein. 2002. *Logistic Regression*. Springer.

[33] Vladimir Kovalenko, Egor Bogomolov, Timofey Bryksin, and Alberto Bacchelli. 2019. PathMiner: A library for mining of path-based representations of code. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 13–17.

[34] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2018. FixMiner: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791* (2018).

[35] Quoc V. Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning*. JMLR.org, 1188–1196. http://proceedings.mlr.press/v32/le14.html.

[36] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 524–535. https://doi.org/10.1109/ICSE.2019.00064

[37] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 593–604. https://doi.org/10.1145/3106237.3106309

[38] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. 213–224. https://doi.org/10.1109/SANER.2016.76

[39] Xuan Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *Empirical Software Engineering* 23, 5 (2018), 3007–3033. https://doi.org/10.1007/s10664-017-9577-2

[40] 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, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[41] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162

[42] Hang Li. 2011. A short introduction to learning to rank. *IEICE Transactions on Information and Systems* 94, 10 (2011), 1854–1862.

[43] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 162:1–162:30. https://doi.org/10.1145/3360588

[44] Bo Lin, Shangwen Wang, Ming Wen, Zhang Zhang, Hongjun Wu, Yihao Qin, and Xiaoguang Mao. 2020. Understanding the non-repairability factors of automated program repair techniques. In *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 71–80. https://doi.org/10.1109/APSEC51365.2020.00015

[45] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 55–56. https://doi.org/10.1145/3135932.3135941

[46] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Tae-young Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. 2019. Learning to spot and refactor inconsistent method names. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 1–12. https://doi.org/10.1109/ICSE.2019.00019

[47] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F. Bissyandé, and Yves Le Traon. 2018. A closer look at real-world patches. In *Proceedings of the 34th International Conference on Software Maintenance and Evolution*. IEEE, 275–286. https://doi.org/10.1109/ICSME.2018.00037

[48] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! An investigation of fault localization bias in benchmarking automated program repair systems. In *Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 102–113. https://doi.org/10.1109/ICST.2019.00020

[49] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing semantic bugs with fix patterns of static analysis violations. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 456–467. https://doi.org/10.1109/SANER.2019.8667970

[50] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 31–42. https://doi.org/10.1145/3293882.3330577

[51] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 615–627. https://doi.org/10.1145/3377811.3380338

[52] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 118–129. https://doi.org/10.1109/SANER.2018.8330202

[53] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. 2020. Automating just-in-time comment updating. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM.

[54] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 166–178. https://doi.org/10.1145/2786805.2786811

[55] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 702–713. https://doi.org/10.1145/2884781.2884872

[56] Rocío Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. 2021. Commit2vec: Learning distributed representations of code changes. *SN Computer Science* 2, 3 (2021), 1–16.

[57] 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*. 101–114. https://doi.org/10.1145/3395363.3397369

[58] Laurens van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of Machine Learning Research* 9, (Nov 2008), 2579–2605.

[59] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An extensible Java bug benchmark for automatic program repair studies. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 468–478. https://doi.org/10.1109/SANER.2019.8667991

[60] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 235–245.

[61] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 441–444. https://doi.org/10.1145/2931037.2948705

[62] Matias Martinez and Martin Monperrus. 2018. Ultra-large repair search space with automatically mined templates: The Cardumen mode of Astor. In *Proceedings of the 10th International Symposium on Search Based Software Engineering*. Springer, 65–86. https://doi.org/10.1007/978-3-319-99241-9_3

[63] Martin Monperrus. 2018. The living review on automated program repair. In *HAL/Archives-Ouvertes. fr, Technical Report*.

[64] Amirfarhad Nilizadeh, Gary T. Leavens, Xuan-Bach D. Le, Corina S. Pasareanu, and David R. Cok. 2021. Exploring true test overfitting in dynamic automated program repair using formal methods. In *Proceedings of the 14th IEEE International Conference on Software Testing, Verification and Validation*.

[65] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*. 815–816.

[66] Tina R. Patil and Swati Sunil Sherekar. 2013. Performance analysis of Naive Bayes and J48 classification algorithm for data classification. *International Journal of Computer Science and Applications* 6, 2 (2013), 256–261.

[67] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 254–265. https://doi.org/10.1145/2568225.2568254

[68] Yihao Qin, Shangwen Wang, Kui Liu, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2021. On the impact of flaky tests in automated program repair. In *Proceedings of the 28th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 295–306. https://doi.org/10.1109/SANER50967.2021.00035

[69] Reuven Y. Rubinstein. 1999. The cross-entropy method for combinatorial and continuous optimization. *Methodology and Computing in Applied Probability* 1 (1999), 127–190.

[70] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs.jar: A large-scale, diverse dataset of real-world Java bugs. In *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories*. ACM, 10–13. https://doi.org/10.1145/3196398.3196473

[71] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*. ACM, 532–543. https://doi.org/10.1145/2786805.2786825

[72] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. 2018. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 130–140. https://doi.org/10.1109/SANER.2018.8330203

[73] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research* 15 (2014), 1929–1958.

[74] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 727–738.

[75] Joshua B. Tenenbaum and William T. Freeman. 2000. Separating style and content with bilinear models. *Neural Computation* 12, 6 (2000), 1247–1283.

[76] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM.

[77] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*. 5998–6008.

[78] Shangwen Wang, Kui Liu, Bo Lin, Li Li, Jacques Klein, Xiaoguang Mao, and Tegawendé F. Bissyandé. 2021. Beep: Fine-grained Fix Localization by Learning to Predict Buggy Code Elements. arXiv:2111.07739[cs.SE].

[79] Shangwen Wang, Ming Wen, Liqian Chen, Xin Yi, and Xiaoguang Mao. 2019. How different is it between machine-generated and developer-provided patches? An empirical study on the correct patches generated by automated program repair techniques. In *Proceedings of the 13th International Symposium on Empirical Software Engineering and Measurement*. IEEE, 1–12. https://doi.org/10.1109/ESEM.2019.8870172

[80] Shangwen Wang, Ming Wen, Bo Lin, and Xiaoguang Mao. 2021. Lightweight global and local contexts guided method name recommendation with prior knowledge. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 741–753. https://doi.org/10.1145/3468264.3468567

[81] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: How far are we?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 968–980. https://doi.org/10.1145/3324884.3416590

[82] 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*. ACM, 1–11. https://doi.org/10.1145/3180155.3180233

[83] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.

[84] Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. 2021. Peculiar: Smart contract vulnerability detection based on crucial data flow graph and pre-training techniques. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 378–389. https://doi.org/10.1109/ISSRE52982.2021.00047

[85] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 226–236. https://doi.org/10.1145/3092703.3092718

[86] Qi Xin and Steven P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. 660–670. https://doi.org/10.1109/ASE.2017.8115676

[87] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 789–799. https://doi.org/10.1145/3183519.3183540

[88] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 416–426. https://doi.org/10.1109/ICSE.2017.45

[89] 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 Transactions on Software Engineering* 43, 1 (2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[90] Bo Yang and Jinqiu Yang. 2020. Exploring the differences between plausible and correct patches at fine-grained level. In *Proceedings of the 2nd International Workshop on Intelligent Bug Fixing*. IEEE, 1–8. https://doi.org/10.1109/IBF50092.2020.9034821

[91] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 831–841. https://doi.org/10.1145/3106237.3106274

[92] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering* (2021).

[93] He Ye, Matias Martinez, and Martin Monperrus. 2021. Automated patch assessment for program repair at scale. *Empirical Software Engineering* 26, 2 (2021), 1–38.

[94] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering* 23, 5 (2018), 2948–2979. https://doi.org/10.1007/s10664-017-9552-y

[95] Pengcheng Yin, Graham Neubig, Miltiadis Allamanis, Marc Brockschmidt, and Alexander L. Gaunt. 2019. Learning to represent edits. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJl6AjC5F7.

[96] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system. *Empirical Software Engineering* 24, 1 (2019), 33–67. https://doi.org/10.1007/s10664-018-9619-4

[97] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated repair of Java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2874648

[98] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 783–794.

[99] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Pruning dynamic slices with confidence. In *2006 ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[100] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An effective approach for localizing faults using convolutional neural networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 445–455.

[101] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 141–151.

[102] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* (2019).

Received July 2021; revised December 2021; accepted December 2021