Rete: Learning Namespace Representation for Program Repair

Nikhil Parasaram, Earl T. Barr, and Sergey Mechtaev

***Abstract*—A key challenge of automated program repair is finding correct patches in the vast search space of candidate patches. Real-world programs define large namespaces of variables that considerably contributes to the search space explosion. Existing program repair approaches neglect information about the program namespace, which makes them inefficient and increases the chance of test-overfitting. We propose RETE, a new program repair technique, that learns project-independent information about program namespace and uses it to navigate the search space of patches. RETE uses a neural network to extract project-independent information about variable CDU chains, def-use chains augmented with control flow. Then, it ranks patches by jointly ranking variables and the patch templates into which the variables are inserted. We evaluated RETE on 142 bugs extracted from two datasets, ManyBugs and BugsInPy. Our experiments demonstrate that RETE generates six new correct patches that fix bugs that previous tools did not repair, an improvement of 31% and 59% over the existing state of the art.**

***Index Terms*—Program Repair, Deep Learning, Patch Prioritisation, Variable Representation.**

I. INTRODUCTION

Automated program repair struggles to find correct patches in vast search spaces of patches. First, exploring a large search space takes time, so program repair techniques can fail to find a patch because they exceed their time budget. Second, even if a program repair technique finds a patch that passes the test suite, this patch may be incorrect, *i.e.* it may *overfit* the test suite, because test suites incompletely capture specifications.

Every program defines a namespace and uses scoping rules to control variable visibility. Existing program repair algorithms neglect information about program's namespace when searching for repairs, which reduces their effectiveness and increases test-overfitting [1]. Patch generation techniques that use machine learning usually fall into three categories: They (1) ignore information about program variables, effectively only learning to choose the template into which to insert variables [2]; (2) only extract variables from local context [3], [4]; or (3) learn information about program namespace implicitly [5], but have difficulty handling long-range dependencies [6]. At each program location, many visible variables are not local, so tools that fall into the first two categories cannot effectively synthesize patches that require non-local visible variables. CoCoNut [5] falls into the third category, so, in principle, it can learn long-range dependencies, but our experiments (Table IX) show that it fails to generate five correct patches because it prefers variables in the local neighbourhood to more suitable visible variables.

We propose RETE, a new program repair technique, to address this problem. RETE prioritises visible variables in a program namespace by their likelihood of being used at a given program location. In the spirit of neuro-symbolic computation [7], RETE combines program analysis and machine learning to learn rich project specific *semantic* information about the namespace. Specifically, it uses variables' CDU chains, def-use chains augmented with control flow, to learn latent, low-dimensional representations of program variables that capture their semantic affinities.

RETE generates patches by inserting program variables into patch templates. It separates patch template generation and prioritisation into two steps and defines an interface for each step to facilitate the use of different algorithms for each one. Thus, RETE is a framework that integrates existing program repair approaches. RETE prioritises patches by combining the ranks of variables and the ranks of patch templates into which the variables are inserted into a single ranking. Any combination of tools can be used to extract these individual template and variable ranks. We implement three instances of template ranking using existing approaches: (1) the plastic surgery hypothesis [8] that extracts templates from the buggy program, mutates them, and prioritises the mutated templates based on the syntactic distance from the donor code; (2) Prophet's enumerative synthesiser and machine learning based prioritiser [1]; and (3) Trident's constraints-based synthesiser [9] and combine them with our various variable ranking algorithms (Section III).

We implemented RETE for C and Python. We chose C because of C's importance and because many program repair techniques have targeted it. We target Python because of its ever-increasing importance and the fact that its default dynamic typing heightens the importance of namespace information, since without static types, every visible variable is a candidate for fixing a bug.

To evaluate RETE, we use two benchmarks: 107 bugs extracted from BugsInPy [10] and 35 bugs extracted from ManyBugs [11]. We extracted these bugs to match RETE's defect class described in Section III-A (with a restriction to inserting/modifying single line statements). The evaluation demonstrates that RETE generates patches for 29 of our 107 BugsInPy bugs, and generates 8 for our 35 bugs ManyBugs. When we adapted Prophet to work on Python, despite its age, it outperformed the previously available state of the art, CoCoNut: fixing 21 bugs to CoCoNut's 16 bugs on our dataset (section V). On Python, RETE outperforms Prophet-for-Python by six correct fixes. When using RETE's variable prioritisation, Prophet-for-Python generates 3 more bugs than it does on its own. On the ManyBugs dataset, Trident's constraint-based

algorithm, augmented with RETE's template enumeration and variable prioritisation, generates more correct patches than the state of the art tools SOSRepair [12], Prophet [1] and Trident on their own.

The paper makes the following contributions:

- We introduce the problem of learning program namespace and present two solutions using deep learning and feature engineering which use information from program analysis.
- We present a generic algorithm to integrate variable prioritisation into existing repair techniques, instantiated for the Plastic Surgery Hypothesis, Prophet and Trident.
- We realise these contributions in RETE, and evaluate them on real bugs in C and Python projects.

RETE's implementation, and the scripts and data used to evaluate it, can be found in the accompanying package https://github.com/norhh/Rete

II. OVERVIEW

To repair real-world bugs, program repair has to explore huge search spaces of candidate patches. Consider the developer patch for a bug in Black [13] in Figure 1a. Even if we restrict the number of field accesses with the operator '`.`' to the maximum of two, there are $11\,118$ visible variables and object field accesses at the fix location. A popular approach for reducing the search space, employed by a large number of existing program repair tools, is to ignore the variables that are not local to the fix. However, considering only local variables makes some bugs impossible to repair and increases the likelihood of generating test-overfitting patches. For example, the field access `self.previous_line.is_comment` in Figure 1a, which is used in the fix, does not exist anywhere else in the entire program. Even assuming that a prioritisation algorithm shortlists 100 candidate variables and we know the exact fix template given in Figure 1b, constructing such a patch would require examining 4950 possibilities to fill the variables into the template.

Program repair approaches relying on program synthesis struggle to generate patches for programs with large namespaces. For example, Trident [9] enumerates patches using patch templates from its search space and checks whether they satisfy the specification constraints. Applied to the bug in Figure 1, for simplicity, let's assume that Trident uses $\mathbf{T}$ patch templates, Trident must enumerate *ca.* 8–9 orders of magnitude ($\mathbf{T} \times 11118^2$) patches to find this fix, a clearly infeasible number of patches. Prophet [2] and SPR [14] map variables to the values that they hold during test execution. They use this mapping to instantiate templates. If the number of candidate variables are large, this can lead to test-overfitting, since many variables can take the same values during test execution, leaving these tools unable to differentiate them.

RETE efficiently synthesises the correct patch in Figure 1a by prioritising the correct patch in the first 1000 patches. To do this, RETE employs a novel method to rank variables using a project-independent representation of a program's namespace.

A. RETE's Template Generation

RETE's core contribution, learning program namespaces, is orthogonal to existing program repair algorithms. Thus, RETE provides an extensible framework for program repair, which seamlessly integrates existing patch generation algorithms. We consider three archetypal algorithms: the plastic surgery hypothesis [8], Prophet's enumerative synthesiser [1], and Trident's constraints-based synthesiser [9].

We now show how RETE uses the plastic surgery hypothesis. Our interpretation of the plastic surgery approach starts with existing program statements as partial patches and edits them using one of three operations: replacing a variable with a hole $\square$, appending an operator with holes for its operands, or removing an operator and its operands. The synthesis algorithm searches for a minimal edit patch. The cost of adding or removing an operator equals the number of holes added or removed. Consider the statement $x + a \times b$, the set $\{x+y, a+self.W\}$, and the rewrite chain $x+y \rightarrow_1 x+\square_1 \rightarrow_2 x+\square_1 \times \square_2 \rightarrow_3 x+a\times\square_2 \rightarrow_4 x+a\times b$. The chain shows that $x+a\times b$ is 2 edits from the set because the cost of both $\rightarrow_1$ and $\rightarrow_2$ is 1 and the cost of both $\rightarrow_3$ and $\rightarrow_4$ is 0. Section III-D explains how we set the edit costs. RETE later fills the variable holes with concrete variables using the ranking it learns.

In Figure 1c, the code block in lines 1-5 (The if block with the comment) is two edits away from the required patch in Figure 1c:

```
1.self.pl              → is_decorator
2.self.pl.is_decorator → self.pl.is_comment
where self.pl represents self.previous_line
```

This patch is correct, if `self.previous_line` is not `None`[1]. As is common practice in APR, RETE guards such deferences with NoneType checks. Given these edits and its NoneType heuristic, RETE generates the human-equivalent patch as shown in Figure 1a.

B. RETE's Variable Prioritisation

RETE's variable ranking model shortlists the most likely variables at the location of the fix based on a partial patch. This reduces not only the number of variables to examine but also the likelihood of test-overfitting. In Figure 1c, the correct variables to fill the holes in "$\square^1$ `and self.previous_line.`$\square^2$" are ranked 4^{th} and 5^{th}, much less than 782 and 12, the number of visible variables at each hole. The rank for the second variable is not as good as that of the first since no identifier that could fix the problem appeared in a similar context sufficiently often for the model to learn it. Purely ML-based approaches, *e.g.* those using Neural Machine Translation (CoCoNut), struggle to fix such bugs, since the field, `is_comment` is too sparse for the model to learn to associate it with `self.previous_line`.

III. RETE

The central "signal" hypothesis of this project is that variables with similar semantics are used in similar ways across

[1]NoneType is Python's unit type.

```
if (
      is_decorator
      and self.previous_line
):
    return 0, 0

+ if (
+     is_decorator
+     and self.previous_line
+     and self.previous_line.is_comment
+ ):
+    return 0, 0


newlines = 2
if current_line.depth:
    newlines -= 1
```

(a) The developer fix for a bug in Black.

```
if (
    is_decorator
    and self.previous_line
):
    return 0, 0


if □¹ and □²:

    return 0, 0


newlines = 2
if current_line.depth:
    newlines -= 1
```

(b) A template for a hypothetical repair tool.

```
if (
    is_decorator
    and self.previous_line
): # Code block used to generate fix
    return 0, 0


if □¹ and self.previous_line.□²:

    return 0, 0


newlines = 2
if current_line.depth:
    newlines -= 1
```

(c) Edited locations with abstract variables.

Fig. 1: A bug in Black Python formatter, and how it can be fixed with patch templates.

code bases; this *usage signal* complements and can supersede signal in raw lexical similarity of names. We introduce conditional def-use chains to capture usage signal.

We consider a C-like programming language $\mathcal{L}$. Program $p \in \mathcal{L}$ is a set of function declarations. In p, let V be its set of variables, and S be the set of all statements in L. Let $\square$ represent a missing variable and $V_\square = V \cup \{\square\}$. The language $\mathcal{L}_\square$ extends $\mathcal{L}$ by replacing V with $V_\square$. Let $\delta \in \Delta$ be a patch; $\delta(p)$ denotes the application of δ to p. $\Delta_\square \subset \Delta$ is a set of patch templates where $\delta_\square(p) \in \mathcal{L}_\square, \forall \delta_\square \in \Delta_\square$. $\delta_\square(\overline{v})$ denotes instantiating $\delta_\square$ with the variables $\overline{v} \in V^n$, where $|\overline{v}| = n$. RETE's defect class [15] consists of those bugs that instantiations of its patch templates can fix.

A. The Patch Ordering Problem

Generate-and-validate program repair approaches find patches by enumerating and testing a large number of candidate patches. Typically, the first patch found that enables the program to pass the test suite is returned as the solution. Thus, the order in which the patches are enumerated is crucial. First, it affects patch quality since not all patches that pass the tests are correct. Second, it affects the speed of patch generation since it determines the number of test executions required to find a suitable patch.

Problem 1 (The Patch Ordering Problem): Given a test suite T, and a program p that does not pass T and a set of patches Δ, find the bijection $O : \Delta \to [1..|\Delta|]$ such that

$$\operatorname*{argmax}_O P\Big(O(\delta_i) < O(\delta_j) \mid \delta_i(p) \text{ is correct} \vee \big(\delta_i(p) \text{ is plausible} \wedge \delta_j(p) \text{ is implausible}\big)\Big),$$

where $\delta(p)$ is plausible if it passes all tests in T and implausible otherwise.

O is a patch ordering that simultaneously maximises the probability that correct patches precede plausible patches, and the probability that plausible patches, in turn, precede implausible patches.

RETE solves a restricted version of this general problem: it optimises O only over instantiated templates $\delta_\square(\overline{v})$. Because of this restriction, RETE decomposes the patch ordering problem into two subproblems: that of ordering templates, aka $\delta_\square$, (Section III-B) and, for each template, the problem of finding the correct variables to fill those holes, *a.k.a.* finding $\overline{v}$ (Section III-C).

B. Prioritising Templates via Distance

Let $\alpha[vars(\alpha)/\square]$ be the buggy context with holes replacing all its variables, $vars(\alpha)$. We formulate the problem of finding the best template for a particular bug in the program p as a graph search problem, starting from $\alpha_\square$. Let $G = (\Delta_\square \cup \{\alpha_\square\}, E)$ be our graph. The distance function $d\colon E \longrightarrow \mathbb{W}$ weights each $(\delta^s_\square, \delta^t_\square) = e \in E$. Templates are ordered by their distance from $\alpha_\square$.

Different APR approaches define d differently. Section IV-A gives our definition. Techniques such as DirectFix [16] define d as the minimal number of sub-expression substitutions required to construct the patch from a buggy statement. Techniques such as Prophet [2] order templates using Maximum likelihood estimation, implicitly defining d using probability.

C. Learning Namespace Representations

In this paper, we introduce namespace representations learning, which aims to learn latent, low-dimensional representations of program variables, which preserves semantic properties of variable uses, and thus facilitates such applications as variable prioritisation for patch synthesis.

We now describe how we capture a variable's uses in a Conditional DU (CDU) chain, a new data structure that augments a classical DU chain with control information. A variable may have many CDU chains, so we describe how we sample them before describing how we use them to learn embeddings that capture affinities between variables (their names and uses) and holes, each of which represents a potential variable use.

The predicate $D : S \times V$ indicates if a variable is *defined* in a statement; the predicate $U : S \times V$ indicates if a variable is *used* in a statement. For a variable v and a definition d (*i.e.* $D(d, v)$), we say that d reaches an arbitrary use-statement s (*i.e.* $U(s, v)$), if there exists at least one execution path from d to s along which no other statement $s' \neq s$ satisfies $D(s', v)$. A

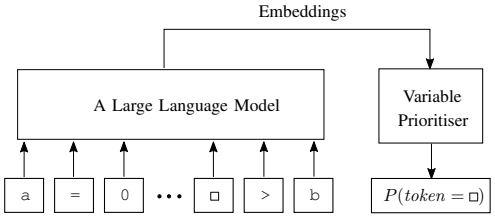

Fig. 2: Processing a sample CDU chain.

def-use chain of the variable v is the sequence of all statements $s_1, ..., s_n$ *s.t.* 1) $D(s_1, v) \wedge \bigwedge_{i=2}^{n} U(s_i, v)$; 2) The definition of v in s_1 reaches s_i for all $i > 1$ along at least one path; and 3) $\forall i, j$ *s.t.* $i < j \wedge i \neq 1$, s_i precedes s_j in the source code.

A node d of a graph *b-dominates* a node e if every path starting from a node b to e traverses d. $Dom_b(d)$ denotes the set of all nodes that *b-dominate* the node d. For the statement s in program p, let $(g_1 \dots g_n)$ be the set of conditional statements in p each of whose elements g_i dominate s. For the sequence $seq = \langle a_1, \dots, a_n \rangle$, we write $x \in seq$ to denote $\exists i \in \{1, \dots, n\}\,.\, x = a_i$.

Definition 1 (Conditional Definition-Use (CDU) Chain):
For the variable v, a *conditional def-use chain w.r.t.* an initial node b in a control flow graph is the sequence of statements $c = \langle s_1, \dots, s_n \rangle$ where

- A subsequence of c is the DU chain d of the variable v; and
- Any statement $s_i \notin d$ is a conditional statement *s.t.* $\forall s_j \in d\,.\, i < j \implies s_i \in Dom_b(s_j)$.

CDU chains are formed by interleaving a DU chain of v with all conditional statements that b-dominate a statement in the underlying subsequence of the DU chain. In c, the condition g_i precedes the arbitrary element s in c's DU chain if g_i dominates s. Definition 1 non-deterministically orders the conditions *w.r.t.* each other, subject to the constraint that a conditional must precede all statements it dominates. Using b-domination allows us to choose a starting node closer to a variable's uses than a program's entry.

Our intuition is that CDU chains of a given length likely have more information about their variable than arbitrary code snippets of the same length. CDU chains are related to program slices, which consist of all the statements and predicates that might affect a set of variables at a program point [17]–[19]. Unlike slices, CDU chains ignore, by design, a variable's data dependencies for two reasons: 1) to avoid the data dependency clusters that bloat many slices [20] and 2) to focus on capturing variable-specific signal.

The *variable prediction task* takes a list of statements with a hole at a variable use and predicts the correct variable to fill that hole. RETE models as a masked language modelling problem. Pre-trained, masked language models for code, based on transformers [21], fine-tuned with a small amount of labelled data, achieve state-of-the-art performance in different software engineering applications [22]–[24]. Accordingly, RETE adopts this approach to learn affinities between variables and their uses in CDU chains.

To train its variable prioritiser model, RETE repeatedly serialises each CDU chain and masks each use of a variable, not just the defined variable, ignoring all other tokens. For instance, three CDU chains containing five variable uses would generate 15 distinct masked training instances. Figure 2 shows us how the model works. For a CDU chain for the variable `a` ending with `a > b`, one instance replaces the final use of `a` with □ to mask it out and then feeds it to a pre-trained large language model to convert the input into a sequence of embeddings. RETE then runs these embeddings through a feed-forward network with a softmax layer to realise the task-specific fine-tuning.

Many CDU chains can traverse a given buggy line, so, while its individual CDU chains may be short, the aggregate length of a variable's set of CDU chains can be large. Transformers take a maximum length sequence of tokens as input. Thus, the serialisation of a variable's CDU chains may need to be compressed. Section IV-B details how RETE's framework serialises and compresses CDU chains.

D. Jointly Prioritising Patches

RETE ranks patches by combining template and variable priorities. RETE uses min priority queue, ordered by Equation (1). Equation (1) defines $score(\delta_{\square}(\overline{v}))$ using $td(\delta_{\square}) \in \mathbb{N}$ which gives $\delta_{\square}$, the priority of the template and $P(\delta_{\square}, \square, v) \in [0, 1]$, the probability of $v \in V$ being the candidate to fill the single hole □. Template distance td (Section III-B) calculates the distance of the template from the source node extracted from the localiser (*i.e.* $td(\delta_{\square}) = d(\delta_{\square}^{S}, \delta_{\square})$). The variable prioritiser described in Section III-C provides variable probabilities. A Lower score means a better patch.

$$score(\delta_{\square}(\overline{v})) = td(\delta_{\square}) + \frac{\theta}{|h(\delta_{\square})|} \sum_{\square_i \in\ h(\delta_{\square})} \frac{1}{P(\delta_{\square}, \square_i, v_i)} \quad (1)$$

where θ is a constant we use to control the growth of the summands, and the function $h(\delta_{\square})$ returns the set of holes in $\delta_{\square}$. We use $\frac{1}{P(\delta_{\square}, \square_i, v_i)}$ in the summation because it gives lower score for variables with a higher probability. As the range of $td(\delta_{\square})$ is in $\mathbb{W}$, various template scores differ by integers, the variable prioritisation score starts to matter when $P(\delta_{\square}, \square_i, v_i) < \theta$, as the score breaks the barrier of 1. We use $\theta = 0.073$ as this value performed the best in our experiments. We prioritise patches by separately prioritising templates and variables rather than directly prioritising instantiated patches because the number of instantiated patches is much larger than the number of templates, making it infeasible in practice.

IV. RETE'S IMPLEMENTATION

Figure 3 shows RETE' architecture, which has three main components: (1) Patch Generator, (2) Patch Prioritiser combining Variable Prioritiser and Template Prioritiser, and (3) Patch Checker. RETE's implementation takes in a buggy program p and a suspicious line number and feeds p to its patch prioritiser, which combines its template and variable prioritisers to produce a patch that it checks with its Patch

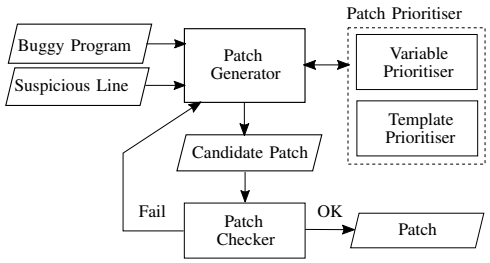

Fig. 3: Architecture of RETE.

Checker. If the check fails, RETE generates and checks patches until it finds a successful patch. RETE implementation is a framework, so its implementation components are not fixed; they can be instantiated with various existing approaches. For instance, its checker can be anything ranging from a test suite to a patch specification extracted from symbolic execution.

When realised as a generate-and-validate approach, RETE prioritises tests that failed the previous run for efficiency in practice. To combine templates and variable priorities, RETE lazily constructs a priority queue ordered by Equation (1).

A. Lazily Prioritising Templates

To realise RETE's template prioritisation scheme (Section III-B), we need to define the buggy context, our templates, and a distance function for our templates, then use these components to build a weighted graph. To define the buggy context, we use fault localisation. Many different localisers have been used in APR for C. For our C dataset, we use the Ochiai statistical fault localisation [25], since it performs well [26]. Variation in localisation has confounded tool comparison in C. Perhaps, for this reason, prior work on Java APR [27], [28] assumes perfect localisation, as it facilitates tool comparison. CoCoNut is the only previous Python APR work of which we are aware, and it assumes perfect localisation. For our Python dataset, we follow its lead.

We turn to the confirmation of the Plastic Surgery Hypothesis (PSH) [8] to define our templates. PSH shows that many fixes can be constructed from existing code in a codebase. Relying on the PSH, we construct templates starting from atomic statements surrounding the buggy context. We templatise these statements by replacing their variables with holes, one at a time. A statement with three variable uses would generate three distinct templates, each with a single hole. We start with only single-holed templates to preserve signal in the names of the other variables in keeping with RETE's central conceit: the application of Firth's dictum [29] to variables.

The distance between two templates is the difference in the number of holes each has. Thus, we prefer templates with fewer holes, in keeping with PSH, are likely to generate instantiations closer to the buggy statement in edit distance.

Starting from this initial set of templates, RETE's framework uses Dijkstra's algorithm [30] to lazily construct its template graph. First, the framework interconnects the initial templates with zero cost edges to favour their use. It weights subsequent edges by the distance of the templates they connect. It moves to the next template only when it has enumerated all variable instantiations of the current template whose score, Equation (1), exceeds the distance of the next template. RETE's framework constructs new templates from existing ones by replacing, appending, or removing statements. Replacing a statement cannot change the number of holes: it must both fill a hole and replace a variable with a hole. Append and remove change the number of holes, so long as the resulting template has at least one hole, *i.e.* remains a template. When filling a hole, we try each of the variable prioritiser's top 30 variables. This restriction was because maintaining all possible bindings is quite expensive, and we observed that it was unnecessary in practice on our corpus. We hypothesize that this observation generalises. We picked 30 as it effectively balanced cost and performance on our corpus. When creating a template, RETE's framework nondeterministically chooses an operation. We chose the templates surrounding the line of code as the set of initial templates, and we restricted the template size to a single statement and the size of the set to 20 templates.

B. Variable Prioritisation with CodeBert

We instantiate the pre-trained component of RETE's variable prioritiser (VP) with CodeBERT [31] and fine-tuned CodeBert for C and Python datasets. Since CodeBert is not trained for C, we replaced C-specific symbols, such as $NULL$ with 0, and dropped $volatile$ quantifiers. We then feed this modified CDU chain to CodeBert to produce embeddings for the task-specific model that fine-tunes CodeBert to RETE's variable prioritisation task: to accurately predict variables for holes in CDU chains. VP's task-specific fine-tuned subcomponent is a feed-forward layer with a softmax function, implemented using Huggingface's open-source transformer [32].

At inference time, we have a buggy statement with a hole. We gather the CDU chains that share this statement, as they comprise the variables that can fill the hole. Under preliminary experimentation, concatenating these chains was inaccurate, so we decided to build a query by interleaving them. To do so, we order statements by reachability, then line number. If one statement is reachable from another in the program's control flow graph, the statement that is reached follows the other in the interleaving; if the statements are mutually reachable or unreachable, we order them by line number. We perform these actions to maximise variable diversity, as we want to improve the cases where uncommon variables are used to fix the bug by increasing unique variable information in our CDU chains.

To fine-tune the model, we trained the task-specific sub-model on interleaved CDU chains so that the training data matches the prediction queries. To produce the shared hole needed for interleaving, we converted each input program into a set of single-holed programs, each with a different variable replaced with a hole. We interleaved the set of CDU chains that pass through each single-holed program's holed statement.

CodeBERT's window size is 512 tokens. In our corpus, 5692 CDU chains pass through a buggy line on average, each having an average length of 34. Naïvely interleaving these produces

Feature	Description
`lvalue`	count of variable definitions
`rvalue`	count of variable uses
`for_init`	count of **`for`** loop initialisation uses
`for_cond`	count of **`for`** loop condition uses
`for_lcv`	count of loop control uses
`while_cond`	count of **`while`** loop condition uses
`if_cond`	count of **`if`** condition uses
`hole_to_def`	distance between hole and the def
`last_use`	distance to last use
`hole_window`	count of uses in k lines around hole
`operator_histo`	multiset of counts of operator/function uses
`is_global`	local/global variables

TABLE I: The features that our random forest uses. Each feature is tracked for each variable in scope.

inputs whose average length is $5692 \times 34 > 512$. To cope with this mismatch, we compress our inputs. First, we consider only cardinality at most five subsets of the CDU chains that pass through a single-holed statement. Then pick that subset whose chains maximise the number of distinct variables that they use, to maximise, as with dropping duplicates above, to increase diverse information on variables. This is the NP hard unweighted maximum coverage problem [33], whose solution we greedily approximate [34]. If required, we next reduce the number of conditions in each chain in the interleaving to the two that use the most variables. If the input is still too long, we truncate it.

a) Random Forest Variable Prioritiser: Several existing program repair techniques [4], [35] use machine learning with feature engineering to tackle test-overfitting. These techniques include features related to program variables, some of which are language-specific and not applicable to C or Python. In the spirit of these techniques, we implemented an alternative approach, based on feature engineering. This approach uses a random forest, and the language-independent features in Table I, to rank program variables. We train this random forest using Scikit-learn [36].

V. Evaluation

This evaluation uses many configurations of APR techniques, so it starts by analytically defining APR components, then using those components to establish a new taxonomy. It then turns to describing experiments whose aim is to answer the following questions:

- Do our conditional definition-use (CDU) chains contain strong signal about variable usage? (Section V-B)
- Is RETE's prioritisation strategy effective? (Section V-C)
- Does the best combination of RETE's components advance the state of the art? (Section V-D)

a) Corpus: Our corpus consists of three bug datasets whose bugs belong to RETE's defect class (Section III-A), as shown in Table II. The programs in P28 were sampled uniformly from GitHub on 21/10/21. We exclude two samples namely wireshark-37122-37123 and gzip-3fe0caeada-39a362ae9d from MB37 used in Trident [9], since we could build them along with our ML libraries. MB35 and BG107 include a test suite; the number of tests averages 62.8 for BG107 and 91.3 for MB35. P28 does not need one as we used it to train and evaluate RETE's neural variable ranker.

BG107	107 bugs from the BugsInPy [10] dataset.
P28	28 Python programs with artificially added holes.
MB35	The ManyBugs [11] subset used in Trident [9].

TABLE II: Evaluation corpus of bugs in RETE's defect class.

	Validator	
$\mathcal{V}$	Validation against Tests	-
$\mathcal{T}$	Trident's patch Specification	[9]
$\mathcal{A}$	Angelix's patch Specification	[38]
$\mathcal{S}$	SOSRepair's patch Specification	[12]
	Patch Generator	
A	Angelix's Angelic Values	[38]
S	SOSRepair's Database of Snippets	[12]
P	SPR's Transformation Schemas	[14]
C	CoCoNut's Neural Machine Translation	[5]
E	Trident's Naïve template enumeration	[9]
G	GenProg's Genetic Algorithm	[39]
S	Plastic Surgery Algorithm	[8]
	Patch Prioritisation	
D	DirectFix's modification minimisation	[16]
C	CoCoNut's Neural Machine Translation	[5]
E	Trident's expression size minimisation	[9]
T	Trident's side effects minimisation	[9]
P	Prophet's Maximum Likelihood Estimation	[1]
S	Plastic Surgery Algorithm	[8]
	Variable Prioritisation	
E	Naïve variable enumeration	-
H	Heuristic discussed in Section V-B	-
B	CodeBERT	[31]
G	GraphCodeBERT	[40]
D	CodeBERT fine-tuned on DU chains	Section III-C
C	CodeBERT fine-tuned on CDU chains	Section III-C
F	Random Forest	Section V-B

TABLE III: Program Repair Components Used in Evaluation.

b) Experimental Setup: We split the test suite into two parts (20-80), taking care that the smaller part includes at least one failing test. The smaller part is sent to a subject APR tool for patch generation. A patch is plausible if it passes the test suite. We consider a patch correct if it is plausible, and manual inspection deems it equivalent to the patch written by the developer. All the patches generated and used in this evaluation are available in the reproduction package.

For all training and fine-tuning, we used the buggy datasets with a split of 90-10 for training and testing. All the hyperparameters are tuned by using K-fold cross-validation with $k = 5$ and grid search. No layers were frozen, since we had a large enough dataset and a set of low learning rates during the grid search. We use Adam optimiser with a weight decay fix [37].

We conducted experiments inside a Docker container on a CPU of 2.7 GHz machine running on Ubuntu 21.04 with 16GB of memory and Geforce RTX 3070M. We used 4 hours timeout for each tool.

A. Tool Configurations and Baselines

RETE's key contribution is in the variable prioritisation, which is designed to improve patch prioritisation for program

Configuration	Tool	Reference
$\mathcal{V}_{\text{P}}^{\text{P}}$	Prophet	[1]
$\mathcal{T}_{\text{T}}^{\text{E}}$	Trident	[9]
$\mathcal{A}_{\text{D}}^{\text{A}}$	Angelix	[38]
$\mathcal{S}^{\text{S}}$	SOSRepair	[12]
$\mathcal{V}_{\text{C}}^{\text{C}}$	CoCoNut	[31]
$\mathcal{V}^{\text{G}}$	GenProg	[39]

TABLE IV: Configurations of standard tools.

repair, and is orthogonal to existing patch generation and prioritisation techniques. To evaluate its impact in isolation, we dissect existing program repair techniques into interchangeable parts, and consider their combinations with RETE's approach. This analysis differs from the existing high-level classification of APR tools [41] into heuristic-based, constraint-based, and learning-based since our goal was to abstract over irrelevant components of existing tools that would complicate the objective evaluation of the proposed technique.

We analytically divided repair techniques into three main parts: patch generation, patch checking and patch prioritisation. Patch generation explores the space of patches by enumeration [14], meta-heuristics [39], neural machine translation [5], [42], or SMT-based program synthesis [43]. Patch checking determines whether a candidate patch meets the correctness criteria, either via validation against a test suite [39] or verification, as by solving constraints [9]. Patch prioritisation ranks patches by their likely correctness, as with syntactic distance [16] or machine learning [2]. We denote such techniques as $\mathcal{X}_b^a$, where $\mathcal{X}$ is a validator, a is a patch generator, and b is a patch prioritiser. For RETE, we further split patch prioritisation into template and variable prioritisation, and combine them using the technique described in Section III-D. Such configurations are denoted by the notation $\mathcal{X}_{R(b,c)}^a$, where b denotes a template prioritisation technique, and c denotes a variable prioritisation technique.

The considered components are tabulated in Table III. Based on this notation, Angelix [38] is $\mathcal{A}_{\text{D}}^{\text{A}}$ since it uses its own patch specification and synthesis methods while using DirectFix [16] patch prioritisation. Angelix uses logical constraints that encode program semantics and information extracted from tests, and validates candidate patches against these constraints; we refer to this validation method as A. Trident is denoted by $\mathcal{T}_{\text{T}}^{\text{E}}$ since it uses different constraints that encode information about side effects. Finally, Prophet is $\mathcal{V}_{\text{P}}^{\text{P}}$ since it extends SPR [14] using an ML-based patch prioritiser. All techniques used as baselines are tabulated in Section V-A.

All the techniques in Table III, except for CoCoNut ($\mathcal{V}_{\text{C}}^{\text{C}}$), are implemented for C. In order to make a more objective comparison, we ported Prophet to Python and trained it on Python programs in the configuration $\mathcal{V}_{\text{R(P, C)}}^{\text{P}}$. Prophet orders partially/fully instantiated patches using machine learning and concretising the partially instantiated patches using SPR's condition synthesis algorithm [14]. In SPR's algorithm, when multiple possible candidate variables have the same potential value map, we improve Prophet by supplanting it with RETE's variable prioritisation to prioritise variables to satisfy the condition. $\mathcal{V}_{\text{R(P, C)}}^{\text{P}}$ differs from $\mathcal{V}_{\text{P}}^{\text{P}}$ in cases where multiple variables satisfy the conditions, $\mathcal{V}_{\text{R(P, C)}}^{\text{P}}$ uses its variable prioritiser (*i.e.* C) to order these unordered candidate variables. Section VI discusses why we consider these baselines. We restrict the defect class to inserting or modifying a single atomic statement for all tools that we ran in the evaluation. This does not cover instances such as inserting for/while block, sequence of atomic statements, method declaration, *etc.* We used this constraint to make the patch generation more feasible by reducing the search space.

B. CDU Chains Contain Strong Signal

To demonstrate the variable usage signal CDU chains contain, we compare token predictors against CodeBert fine-tuned on CDU chains on our variable prediction task.

We use four variable prioritisation baselines — H, F, B, and G — introduced in Table III. The first, H, ranks candidate variables based on their frequency in a program. The second, F, is the random forest implemented using explicit feature engineering; outperforming this technique means that our neural approach is outperforming manual feature engineering. More information on the features used by the random forest available in our reproduction package. These two baselines frame the results; we use the two neural baselines — CodeBert B and GraphCodeBert G — without fine-tuning to demonstrate variable signal in CDU chains. We did not fine-tune GraphCodeBert with CDU chains because it takes both source code and that code's dataflow as input. The difference between the performance of CodeBert and GraphCodeBert in Table V do not appear to warrant the engineering required to extract the dataflow GraphCodeBert needs. We note that CDU chains, by construction, implicitly contain dataflow information, which a neural network trained using them may learn and exploit.

Internally, all of these approaches rank candidate variables, so it is natural to compare them using standard rank measures. Unfortunately, none works well in our setting. We do not use mean average precision (MAP) or normalised discounted gain (NDCG) because we only care about the rank of the first plausible patch, not their density in a prefix of the complete ranking. We do not use Mean reciprocal rank (MRR) because it does not account for search space reduction. We rank the candidate variables *w.r.t.* to the number of variables in-scope, which differs for different samples. MRR does not distinguish cases such as ranking 6^{th} out of 6 vs. 6^{th} out of 1000 variables in-scope.

Instead, we introduce a new measure, which we call the *fraction searched* measure. This measure returns the length of the prefix of ranked variables that must be checked over the total number of candidates, *i.e.* the fraction of the variable search space we had to check. For example, assume we have a search space of 100 variables and, under a particular ranker, the 16^{th} variable is the first that can fill a hole. In this case, the ranker reduced the search space to 16%. Formally, let $p \in \mathcal{L}$ be a program and $p_\square \in \mathcal{L}_\square$ be p with some of its variables replaced with holes. Let $r(\square_i, p_\square)$ be the rank that the variable

TABLE V: The performance of variable prioritisation techniques using the fraction searched measure. The best configuration C, CodeBert fine-tuned with CDU chains, is bolded.

Dataset	samples	H	F	B	G	D	**C**
Python	802k	0.476	0.039	0.232	0.204	0.18	**0.033**
C	652k	0.416	0.032	0.260	0.252	0.07	**0.028**

```
-  if (tif->tif_rawcc > 0 && tif->tif_rawcc != orig_rawcc
+  if ((tif->tif_rawcc > 0)
       && (tif->tif_flags & TIFF_BEENWRITING) != 0
       && !TIFFFlushData1(tif))
     {
```

Fig. 4: Dev. patch for libtiff-2007-11-02-371336d-s865f7b2.

ranker r assigns to the ground truth variable in p that fills $\square_i$ in $p_\square$. The r's fraction searched is

$$F(r, p, p_\square) = \frac{1}{|h(p_\square)|} \sum_{\square_i \in h(p_\square)} \frac{r(\square_i, p_\square)}{|var(\square_i, p_\square)|}$$

where $h(p_\square)$ counts the holes in $p_\square$ and $var(\square_i, p_\square) \subseteq V$ denotes the set of all the variables in-scope $\square_i$ in $p_\square$.

To compute $var(\square, p_\square)$, we consider only variables whose type matches the hole's type. For Python, this does not help much since everything is an object and many objects seamlessly slot into many expressions because of default functions, like `__bool__()`. Hence, we restrict rankings to a fixed vocabulary of variables that varies by bug. We construct this vocabulary by limiting the accessors (".") to 2 to leverage the Law of Demeter [44] and avoid variable explosion.

Table V shows that CodeBERT [31], fine-tuned with CDU Chains, (Column C) outperforms the baselines. In particular, it outperforms the neural baselines — vanilla CodeBERT (B) and GraphCodeBert (G) — by an order of magnitude. We observe that CodeBERT, when fine-tuned on CDU chains, (Column C), performs better when compared to its counterpart fine-tuned on DU chains (Column D). Using this, we can conclude that extending DU chains with conditions helps improve performance. As expected, we observe that feature engineering allows random forest (Column F) to outperform vanilla CodeBert and GraphCodeBert, which are not benefiting from CDU chains. These results are strong evidence that CDU chains are valuable source of signal for the variable prediction task. We find:

> CDU chains contain strong signal: CodeBert fine-tuned on CDU chains (Column C in Table V) outperforms the neural baselines by an order of magnitude; its performance edge over random forest (F) means fewer expensive checks of candidate patches.

Although CDU chains permit CodeBert to outperform our random forest baseline, its inference is expensive. It takes 0.65s on average to respond to queries, compared to *ca.* 0.004s for our random forest.

TABLE VI: On BG107, variable and template prioritisation perform better. The best configuration is bolded.

Dataset	Samples	Correct				Plausible			
		$\mathcal{V}^{E}_{R(E,\,F)}$	$\mathcal{V}^{S}_{R(S,\,H)}$	$\mathcal{V}^{S}_{R(S,\,F)}$	$\mathcal{V}^{S}_{\mathbf{R(S,\,C)}}$	$\mathcal{V}^{E}_{R(E,\,F)}$	$\mathcal{V}^{S}_{R(S,\,H)}$	$\mathcal{V}^{S}_{R(S,\,F)}$	$\mathcal{V}^{S}_{\mathbf{R(S,\,C)}}$
Black	4	0	0	2	2	0	3	3	3
Fastapi	3	0	0	0	0	0	0	0	0
Httpie	3	1	1	2	2	1	2	3	3
Keras	11	0	1	2	2	0	1	7	7
Sanic	1	0	0	0	0	0	0	0	0
Y-DL	3	0	0	1	1	0	0	1	1
Spacy	2	0	0	1	1	0	0	1	1
Tqdm	4	1	1	2	2	1	1	3	3
PySnooper	1	0	1	1	1	0	1	2	2
Tornado	6	0	0	1	2	0	0	2	2
Matplotlib	8	0	2	2	2	0	2	3	3
Luigi	15	1	5	7	7	1	7	7	7
Scrapy	10	0	2	2	2	0	2	2	2
Pandas	36	0	2	4	5	0	2	4	5
Overall	107	3	15	27	**29**	3	21	38	**39**

TABLE VII: Average patch ranking for MB35: variable prioritisation indeed helps since $\mathcal{T}^{S}_{R(S,\,C)}$ and $\mathcal{T}^{S}_{R(S,\,F)}$ outperform other tools by a large margin. Ranks that cannot be assessed are marked with "-".

Bug	$\mathcal{T}^{E}_{T}$	$\mathcal{T}^{E}_{R(E,\,F)}$	$\mathcal{T}^{S}_{R(S,\,H)}$	$\mathcal{T}^{S}_{\mathbf{R(S,\,F)}}$	$\mathcal{T}^{S}_{\mathbf{R(S,\,C)}}$
gmp-a1d3d-f17cb	165350	2563	87339	933	843
libtiff-09e82-f2d98	13313	311	14241	552	513
libtiff-764db-2e42d	90471	7655	724	2	7
libtiff-a72cf-0a36d	$\approx 10^{12}$	-	52428627	15924	15842
libtiff-37133-865f7	$\approx 10^{18}$	-	40842	8786	8566
php-70075-5a8c9	87	51	619	890	782
php-e65d3-1d984	90471	36840	110436	5072	5439
php-63673-2adf5	61	7	52	7	3
Average	$\approx 2 \times 10^{17}$	-	6585360	**4021**	**3999**

C. *Effectiveness of* RETE*'s Prioritisation*

To understand the effectiveness of RETE's patch prioritisation, we compare component combinations for Python and then C. Below, PSH refers to the plastic surgery hypothesis and VP to variable prioritiser. For Python, we use BG107 and these component combinations:

- $\mathcal{V}^{S}_{R(S,\,C)}$ PSH with CDU chain VP
- $\mathcal{V}^{S}_{R(S,\,F)}$ PSH with Random Forest VP
- $\mathcal{V}^{S}_{R(S,\,H)}$ PSH with Heuristic VP
- $\mathcal{V}^{E}_{R(E,\,F)}$ Naïve enumeration with Random Forest VP

Table VI shows that $\mathcal{V}^{S}_{R(S,\,C)}$ significantly outperforms all the other combinations. Combining PSH and RETE's VP does indeed improve patch synthesis since $\mathcal{V}^{S}_{R(S,\,C)}$ and $\mathcal{V}^{S}_{R(S,\,F)}$ both significantly outperform $\mathcal{V}^{S}_{R(S,\,H)}$ and $\mathcal{V}^{E}_{R(E,\,F)}$. $\mathcal{V}^{E}_{R(E,\,F)}$ could fix only three bugs, whereas $\mathcal{V}^{S}_{R(S,\,F)}$ fixes 27 and $\mathcal{V}^{S}_{R(S,\,H)}$ fixes 15. This is because most patches are closer to pre-existing code located somewhere in the buggy program's code [8]; hence, when $\mathcal{V}^{E}_{R(E,\,F)}$ tries to construct a patch in such cases, it faces the harder task of doing so from the ground up without guidance. $\mathcal{V}^{S}_{R(S,\,F)}$ fixes more bugs than $\mathcal{V}^{S}_{R(S,\,H)}$, since it enumerates patches more intelligently (Section III-C). $\mathcal{V}^{S}_{R(S,\,C)}$ fix two more correct bugs when compared against $\mathcal{V}^{S}_{R(S,\,F)}$ since CDUs identify relevant variables better than our random forest baseline.

TABLE VIII: This table compares Prophet and CoCoNut against RETE's best variant ($\mathcal{V}^{S}_{R(S,\,C)}$) and Prophet enhanced with RETE ($\mathcal{V}^{P}_{R(P,\,C)}$). For correct and plausible patches, RETE outperforms the other baselines by generating 7 and 13 additional correct patches against Prophet and CoCoNut, respectively. All the RETE variants are bolded.

Dataset	Samples	Correct				Plausible			
		$\mathcal{V}^{P}_{P}$	$\mathcal{V}^{P}_{R(P,\,C)}$	$\mathcal{V}^{C}_{C}$	$\mathcal{V}^{S}_{R(S,\,C)}$	$\mathcal{V}^{P}_{P}$	$\mathcal{V}^{P}_{R(P,\,C)}$	$\mathcal{V}^{C}_{C}$	$\mathcal{V}^{S}_{R(S,\,C)}$
Black	4	1	2	1	2	3	3	3	3
FastApi	3	0	0	0	0	0	0	0	0
Httpie	3	1	2	1	2	3	3	2	3
Keras	11	2	2	1	2	4	4	3	7
Sanic	1	0	0	0	0	0	0	0	0
Y-DL	3	0	0	0	1	0	0	0	1
Spacy	2	1	1	0	1	1	1	1	1
Tqdm	4	1	2	1	2	2	2	1	3
PySnooper	1	1	1	1	1	1	1	1	2
Tornado	6	0	0	0	2	0	0	0	2
Matplotlib	8	2	2	2	2	2	2	2	3
Luigi	15	8	8	5	7	8	8	8	7
Scrapy	10	2	2	1	2	2	2	3	2
Pandas	36	3	3	3	5	5	5	6	5
Overall	107	22	**25**	16	**29**	31	**31**	30	**39**

TABLE IX: The number of patches that require non-local variables. A variable is considered non-local if it is not directly used in the method. All RETE configurations are bolded. Using RETE with variable prioritisation helps generate non-local variables.

	$\mathcal{V}^{C}_{C}$	$\mathcal{V}^{P}_{P}$	$\mathcal{V}^{P}_{R(P,\,C)}$	$\mathcal{V}^{S}_{R(S,\,C)}$
Correct Patches	16	22	25	29
Correct Patches with non-local Variables	0	0	**1**	**5**

For C, we used the eight bugs in intersection of MB35 and the subset of RETE 's defect class on which its best configuration produces a fix. We use Trident [9] as a validator: C compiles slowly, and Trident's efficient patch specification inference obviates many compilations. We compare these five component combinations: $\mathcal{T}^{E}_{T}$ (vanilla Trident), $\mathcal{T}^{E}_{R(E,\,F)}$, $\mathcal{T}^{S}_{R(S,\,H)}$, $\mathcal{T}^{S}_{R(S,\,F)}$ and $\mathcal{T}^{S}_{R(S,\,C)}$. Table VII shows the results. In some cases, the ranks of patches from $\mathcal{T}^{S}_{R(S,\,F)}$ and $\mathcal{T}^{S}_{R(S,\,C)}$ are extremely low compared to those $\mathcal{T}^{E}_{T}$'s patches. This is because $\mathcal{T}^{E}_{T}$ constructs statements from ground up, hence bugs such as `libtiff-a72cf60-0a36d7f` and `libtiff-371336d-865f7b2` (Figure 4) cannot be synthesized by $\mathcal{T}^{E}_{T}$ as they require constructing a fresh expression with 8-13 different variables and operators. In such cases, we resorted to estimating $\mathcal{T}^{E}_{T}$'s rank on them by assuming that candidate variables were uniformly ordered. Unfortunately, we could not estimate $\mathcal{T}^{E}_{R(E,\,F)}$'s performance the same way, since it uses F as its VP. On average, the search space is reduced by a few orders of magnitude on these seven bugs. The rankings of configurations $\mathcal{T}^{S}_{R(S,\,H)}$ and $\mathcal{T}^{S}_{R(S,\,C)}$ affirm that both RETE's template and variable prioritisers play a pivotal role in synthesising patches.

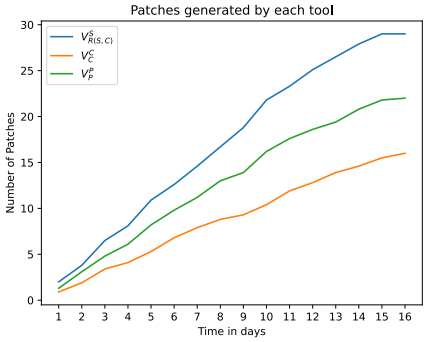

Fig. 5: Patches generated per unit time by RETE ($\mathcal{V}^{S}_{R(S,\,C)}$), Prophet ($\mathcal{V}^{P}_{P}$) and CoCoNut ($\mathcal{V}^{C}_{C}$).

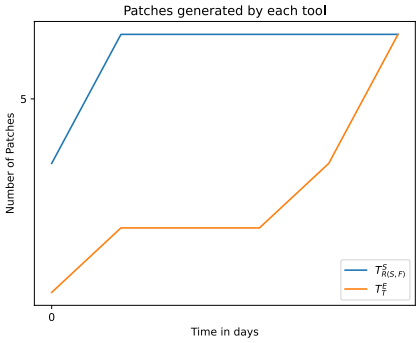

Fig. 6: The number of patches generated by $\mathcal{T}^{E}_{T}$ and $\mathcal{T}^{S}_{R(S,\,F)}$ per unit time: $\mathcal{T}^{S}_{R(S,\,F)}$ generates patches faster than $\mathcal{T}^{E}_{T}$.

> On both our Python and C datasets, enhancing other techniques with RETE's template and variable prioritisers improves their performance.

D. RETE*'s Performance*

We now show that RETE advances the state of the art in 1) time to generate patches and 2) number of correct patches. We close by showing how it speeds Trident, the state of the art C APR tool for handling side effects.

Figure 5 shows how many patches RETE ($\mathcal{V}^{S}_{R(S,\,C)}$), Prophet ($\mathcal{V}^{P}_{P}$) and CoCoNut ($\mathcal{V}^{C}_{C}$) produce per unit time. All 107 programs from the dataset are uniformly ordered and then each tool's execution time to patch each bug, with a timeout of four hours, is summed, upto the time budget. For each budget, we repeat this process 10 times with 10 different orders and average the results. At all budgets, RETE ($\mathcal{V}^{S}_{R(S,\,C)}$) outperforms Prophet and CoCoNut.

Table VIII compares RETE against Prophet [2] and CoCoNut [31] in terms of the number of correct and plausible patches each generates. All RETE variants are bolded in the table. Prophet ($\mathcal{V}^{P}_{P}$) generates 22 and CoCoNut ($\mathcal{V}^{C}_{C}$) 16 correct patches. RETE's best configuration ($\mathcal{V}^{S}_{R(S,\,C)}$) generates 29 correct patches. On Python, RETE improves the state of the art by 31% vs. Prophet and 59% vs. CoCoNut. Prophet, enhanced with RETE's variable prioritisation ($\mathcal{V}^{P}_{R(P,\,C)}$), fixes three more

TABLE X: Plausible and correct patches for bugs in MB35 (C dataset). RETE integrated with Trident, the column labelled $\mathcal{T}^{S}_{R(S,F)}$, generates 18 plausible patches out of 35, of which 8 are correct. The next largest total, GenProg, the column labelled $\mathcal{V}^{G}$, generates 19, of which only two are correct. No other tool configuration generates more correct patches than $\mathcal{T}^{S}_{R(S,F)}$.

Bug	kLoC	Total	Plausible						Correct					
			$\mathcal{T}^{S}_{R(S,F)}$	$\mathcal{T}^{E}_{T}$	$\mathcal{A}^{A}_{D}$	$\mathcal{V}^{P}_{P}$	$\mathcal{S}^{S}$	$\mathcal{V}^{G}$	$\mathcal{T}^{S}_{R(S,F)}$	$\mathcal{T}^{E}_{T}$	$\mathcal{A}^{A}_{D}$	$\mathcal{V}^{P}_{P}$	$\mathcal{S}^{S}$	$\mathcal{V}^{G}$
gmp	145	2	1	1	2	1	0	1	0	0	0	0	0	0
gzip	491	3	3	3	3	2	0	1	1	1	1	1	0	0
libtiff	77	10	8	5	5	5	8	7	4	3	2	1	2	2
php	1,099	19	6	6	6	9	4	9	3	3	3	4	3	0
wireshark	2,814	1	1	1	1	1	1	1	0	0	0	0	1	0
Overall		35	**18**	15	17	18	13	19	**8**	6	6	6	6	2

bugs than vanilla Prophet [2]. Interestingly, we observe each of these two tools exclusively fixes some patches, like patches in `Luigi` and `Pandas`. Although $\mathcal{V}^{P}_{R(P,C)}$ generates more correct patches than Prophet, they generate the same number of total patches. This may indicate that Prophet enhanced with Rete's variable prioritisation reduces the overall overfitting of the generated patches.

A variable is non-local if it is not used in the buggy method. For example, if the variable `var` is directly used somewhere in the method, but `var.a` is not, `var` is local and `var.a` is non-local. Under this definition, most in-scope variables in Python are non-local. Table IX shows that integrating variable prioritisation helps synthesise patches that require non-local variables: $\mathcal{V}^{S}_{R(S,C)}$ fixes five bugs that require non-local variables, whereas the other tools struggle to synthesise correct patches that require them.

> $\mathcal{V}^{S}_{R(S,C)}$ fixes five bugs that require non-local variables, whereas other tools struggle to synthesise correct patches that require non-local variables.

Table X compares the performance of RETE-enhanced Trident [9] ($\mathcal{T}^{S}_{R(S,F)}$) against the state of the art. We did not include $\mathcal{T}^{S}_{R(S,C)}$ for C, despite the fact that it generates better rankings, because Trident's validation step is much faster on average than querying CodeBert (0.28 vs. 0.65 seconds). A faster GPU than ours would probably alleviate this issue.

Although $\mathcal{T}^{S}_{R(S,F)}$ does not synthesise new patches relative to the state of the art, it does synthesise more correct patches. Angelix also generated the two correct patches that $\mathcal{T}^{S}_{R(S,F)}$ generated beyond those generated by vanilla Trident. This is due to patch construction. Angelix ($\mathcal{A}^{A}_{D}$) constructs patches by minimally modifying the existing expressions using SMT queries, while RETE's PSH template constructor tries to find patches close to a set of existing statements. One such patch is Figure 4. Here, Angelix and $\mathcal{T}^{S}_{R(S,F)}$ both successfully generate this patch, since it is easy to modify the existing expression to reach the patch by simply dropping the expression `tif->tif_rawcc != orig_rawcc`. Vanilla Trident, in contrast, does not since the patch requires an expression with 11 components, which is infeasible due to combinatoric explosion. To understand the importance of RETE's variable prioritisation more clearly, we ran an experiment by varying the time limits on these seven bugs that $\mathcal{T}^{E}_{T}$ synthesises to compare it against $\mathcal{T}^{S}_{R(S,F)}$. Figure 6 shows that $\mathcal{T}^{S}_{R(S,F)}$ synthesises patches much faster than $\mathcal{T}^{E}_{T}$.

> RETE-enhanced Trident ($\mathcal{T}^{S}_{R(S,F)}$) repairs bugs faster than vanilla Trident ($\mathcal{T}^{E}_{T}$) while fixing two more bugs.

VI. Related Work

Depending on the correctness criterion, we classify program repair techniques into static and test-driven. Static techniques rely on static analysis or formal specification, while test-driven techniques rely on tests. RETE is a new test-driven technique. We divide program repair algorithms into four conceptual parts: patch generation, which explores the space of candidate patches, patch prioritisation which determines which of the two given patches are more likely to be correct, and patch checking that checks the input patch against a given specification. RETE's key contribution is in patch prioritisation.

Patch Generation and Checking: Various patch generation approaches have been proposed: SPR [14] explores the space of patches by enumeration, GenProg [39] uses meta-heuristic search, CoCoNut [5] and Cure [42] use neural machine translation, and techniques like SemFix [43] and Angelix [38] employs SMT-based program synthesis. Program repair typically realises patch checking by 1) testing as in generate-and-validate techniques [39] or by 2) solving constraints as in semantic techniques [9], [12], [43]. RETE is a generic framework that does not impose a fixed patch generation or checking technique. In this work, we evaluate three instantiations of RETE: for the plastic surgery hypothesis [8], for Prophet's enumerative synthesiser [1], and for Trident's constraint-based synthesiser [9]

Patch Prioritisation: Various patch prioritisation techniques have been proposed. DirectFix [16] prioritises smaller changes. Prophet [2] learns a probabilistic model for ranking patches. CapGen [45] uses AST node information to estimate the likelihood of concrete patches. GetaFix [46] uses a hierarchical clustering algorithm that clusters mined fix patterns into general and specific fix patterns and uses code context to choose an appropriate fix pattern. These approaches ignore information about program variables. Several recent techniques [4], [35], although they focus only on variables in the local context, do learn variable-related features and

thus can prioritise variables. In contrast to these techniques, RETE does not require feature engineering and outperforms our language-agnostic feature engineering approach, as shown in Section V.

Deep Learning for Program Repair: Deep learning has been applied for automatically repairing bugs. DeepRepair [47] leverages learned code similarities using recursive autoencoders [48] to select repair ingredients from code fragments that are similar to the buggy code. Several techniques leverage deep learning to directly sort and transform code [5], [49]. CoCoNut [5] is a generate and validate approach that directly generates multiple patches by using an ensemble of context-aware neural network architecture. RewardRepair [50] uses execution data to improve upon patch synthesis. VarCLR [51] uses Recoder [52] synthesises a sequence of edits over directly synthesising the correct program. SequenceR [53] clusters similar variables by directly passing a stream of tokens to an encoder. Although deep learning techniques implicitly learn information about the program namespace, our experiments show that state-of-the-art deep learning based tools have difficulty handling the long-range dependencies problem [6]. RETE addresses this problem by combining deep learning with program analysis in the form of extraction of CDU chains to learn a project-independent representation of the program namespace. We could not use Cure [42] as a baseline since its reproduction package is not currently public. Additionally, more recent tools, such as Recoder [52] and RewardRepair [50] were not compared against in the evaluation since they were implemented on Java.

Test-overfitting in Program Repair: Test-driven techniques are subject to test-overfitting [41], [54], [55], and several techniques have been designed to address this challenge. Apart from the patch prioritisation techniques discussed above, researchers have proposed using pre-defined databases of transformations to increase the chance of generating correct patch [56]–[58], or generating additional tests [59], [60]. RETE is complementary to these techniques.

Variable Representation: There are various techniques used to represent variables, each with its own strengths and weaknesses. Word2vec [61] and its extensions, such as GloVe [62] and FastText [63], are simpler approaches that model variable representations by encoding tokens. However, these techniques may not capture the full complexity and nuance of natural language. To overcome these limitations, more advanced techniques, such as ELMo [64] and BERT [65], use pre-trained language models to achieve a deeper understanding of language [66], [67]. These approaches have been successfully applied in a range of code modification tasks, including suggesting variable names from code contexts [68], rewriting method and class names [69], and automated program repair [4], [35], [47]. They have also been used for type inference from natural language information [70], [71] and detecting bugs [72], [73]. These techniques, while possessing substantial strengths, are nonetheless impeded by sparse data, particularly when learning about global information outside the network's context window. This can result in suboptimal performance in certain code modification tasks. RETE ameliorates this data sparsity issue by using CDU chains, which contain important information that increases the likelihood of incorporating global information into the network's context. This approach is effective, as demonstrated by the results in Table IX, which show that RETE generates more fixes related to global variables when compared with other techniques.

VII. CONCLUSION

Existing program repair approaches neglect information about the program's namespace when searching for repairs, which reduces their effectiveness and increases test overfitting. This work aims to address this problem by augmenting patch prioritisation with information about program namespace. RETE extracts information from CDU chains and uses it for patch prioritisation. Our evaluation shows that RETE can repair real bugs in open-source projects faster compared to state-of-the-art and also finds more correct repairs.

REFERENCES

[1] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 702–713.

[2] ——, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[3] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 416–426.

[4] Y. Xiong and B. Wang, "L2s: A framework for synthesizing the most probable program under a specification," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–45, 2022.

[5] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "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*, 2020, pp. 101–114.

[6] H.-S. Le, A. Allauzen, and F. Yvon, "Measuring the influence of long range dependencies with neural network language models," in *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, 2012, pp. 1–10.

[7] M. K. Sarker, L. Zhou, A. Eberhart, and P. Hitzler, "Neuro-symbolic artificial intelligence: Current trends," 2021. [Online]. Available: https://arxiv.org/abs/2105.05330

[8] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 306–317.

[9] N. Parasaram, E. T. Barr, and S. Mechtaev, "Trident: Controlling side effects in automated program repair," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–1, 2021.

[10] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh *et al.*, "Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1556–1560.

[11] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[12] A. Afzal, M. Motwani, K. Stolee, Y. Brun, and C. Le Goues, "Sosrepair: Expressive semantic search for real-world program repair," *IEEE Transactions on Software Engineering*, 2019.

[13] Łukasz Langa and collaborators, "Black, a python code formatter," https://github.com/psf/black, 2022, accessed: 2022-05-05.

[14] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166–178.

[15] M. Monperrus, "A critical review of" automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 234–242.

[16] S. Mechtaev, J. Yi, and A. Roychoudhury, "Directfix: Looking for simple program repairs," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 448–458.

[17] M. Weiser, "Program slicing," *IEEE Transactions on software engineering*, no. 4, pp. 352–357, 1984.

[18] K. B. Gallagher, "Using program slicing in software maintenance," Ph.D. dissertation, University of Maryland, Baltimore County, 1990.

[19] T. Gyimóthy, A. Beszédes, and I. Forgács, "An efficient relevant slicing method for debugging," in *Software Engineering—ESEC/FSE'99*. Springer, 1999, pp. 303–321.

[20] S. Islam, J. Krinke, D. Binkley, and M. Harman, "Coherent clusters in source code," *Journal of systems and software*, vol. 88, pp. 1–24, 2014.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[22] E. Biswas, M. E. Karabulut, L. Pollock, and K. Vijay-Shanker, "Achieving reliable sentiment analysis in the software engineering domain using bert," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 162–173.

[23] T. Zhang, B. Xu, F. Thung, S. A. Haryono, D. Lo, and L. Jiang, "Sentiment analysis for software engineering: How far can pre-trained transformer models go?" in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 70–80.

[24] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[25] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[26] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 332–347, 2019.

[27] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 102–113.

[28] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 615–627.

[29] J. R. Firth, "A synopsis of linguistic theory, 1930-1955," *Studies in linguistic analysis*, 1957.

[30] E. W. Dijkstra *et al.*, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[31] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. [Online]. Available: https://aclanthology.org/2020.findings-emnlp.139

[32] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Transformers: State-of-the-art natural language processing," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. [Online]. Available: https://www.aclweb.org/anthology/2020.emnlp-demos.6

[33] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[34] D. S. Hochbaum, "Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems," in *Approximation algorithms for NP-hard problems*, 1996, pp. 94–143.

[35] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, "Automated classification of overfitting patches with statically extracted code features," *IEEE Transactions on Software Engineering*, 2021.

[36] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[37] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[38] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[39] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 3–13.

[40] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "Graphcodebert: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[41] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[42] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.

[43] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.

[44] K. Lieberherr and I. Holland, "Assuring good style for object-oriented programs," *IEEE Software*, vol. 6, no. 5, pp. 38–48, 1989.

[45] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 1–11.

[46] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[47] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 479–490.

[48] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 87–98.

[49] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," *arXiv preprint arXiv:2103.06333*, 2021.

[50] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 1506–1518.

[51] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. Le Goues, "Varclr: Variable semantic representation pre-training via contrastive learning," in *44th IEEE/ACM 44th International Conference on Software Engineering*. ACM, 2022, pp. 2327–2339. [Online]. Available: https://doi.org/10.1145/3510003.3510162

[52] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 341–353.

[53] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, 2019.

[54] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? overfitting in automated program repair," in

Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 532–543.

[55] X. B. D. Le, F. Thung, D. Lo, and C. Le Goues, "Overfitting in semantics-based automated program repair," *Empirical Software Engineering*, vol. 23, no. 5, pp. 3007–3033, 2018.

[56] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 802–811.

[57] J. Kim and S. Kim, "Automatic patch generation with context-based change application," *Empirical Software Engineering*, vol. 24, no. 6, pp. 4071–4106, 2019.

[58] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, pp. 1–45, 2020.

[59] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, "Better test cases for better automated program repair," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 831–841.

[60] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 8–18.

[61] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *Advances in neural information processing systems*, vol. 26, 2013.

[62] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[63] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the association for computational linguistics*, vol. 5, pp. 135–146, 2017.

[64] M. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations. arxiv 2018," *arXiv preprint arXiv:1802.05365*, vol. 12, 2018.

[65] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[66] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[67] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. L. Goues, "Varclr: Variable semantic representation pre-training via contrastive learning," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2327–2339.

[68] R. Bavishi, M. Pradel, and K. Sen, "Context2name: A deep learning-based approach to infer natural variable names from usage contexts," *arXiv preprint arXiv:1809.05193*, 2018.

[69] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019.

[70] R. S. Malik, J. Patra, and M. Pradel, "Nl2type: inferring javascript function types from natural language information," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 304–315.

[71] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, "Python probabilistic type inference with natural language support," in *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, pp. 607–618.

[72] M. Pradel and T. R. Gross, "Detecting anomalies in the order of equally-typed method arguments," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, 2011, pp. 232–242.

[73] M. Pradel and K. Sen, "Deepbugs: A learning approach to name-based bug detection," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–25, 2018.