Empirical Software Engineering manuscript No.
(will be inserted by the editor)

A Study of Common Bug Fix Patterns in Rust

Mohammad Robati Shirzad ·
Patrick Lam

Received: date / Accepted: date

Abstract Rust is a relatively new programming language which allows programmers to write programs that have low-level control over resources while still ensuring high-level safety guarantees (for programs written in safe Rust). Rust's ownership framework enables programs to meet these two seemingly-contradictory goals. The Rust compiler's Borrow-Checker component enforces the ownership framework requirements that ensure Rust's safety guarantees.

Rust is popular: as of 2022, it has ranked first, for the seventh consecutive year, in Stack Overflow's annual Developer Survey as the most-loved programming language. The number of Rust developers is growing as the need for faster and safer software increases.

Yet, to our knowledge, no research has sought to identify the most pervasive bug fix patterns within Rust programs. In this project, we introduce Ruxanne, a tool for analyzing and extracting fix patterns in Rust. Ruxanne implements a novel embedding of Rust code into fixed-sized vectors. Using Ruxanne, we mined the top 18 most-starred Rust projects in GitHub to discover the most common bug fix patterns committed to their repositories. We analyzed 87,726 code changes drawn from 57,214 commits across these 18 projects. After clustering the code changes, and conducting a manual analysis, we identified 20 groups of cross-project bug fix patterns, which we categorize as (1) general patterns and (2) borrow-checker-related patterns. Among the general patterns, the most frequently observed pattern is when the user either adds or removes struct fields. In the case of borrow-checker-related patterns, the most common pattern we encountered is when the user removes a clone() call. We describe all detected patterns and their implications to automated program repair.

Competing Interests: We have no competing interests and are funded by a Discovery Grant from Canada's Natural Science and Engineering Research Council.

Mohammad Robati Shirzad — E-mail: mrobatis@uwaterloo.ca
Patrick Lam — E-mail: patrick.lam@uwaterloo.ca
University of Waterloo — Address: 200 University Ave W, Waterloo, ON N2L 3G1

Keywords Bug patterns · Pattern mining · Bug fix changes · Rust

Mathematics Subject Classification (2020) 68N01 · 68N15

1 Introduction

Rust is a relatively new programming language (Klabnik and Nichols, 2019) which allows programmers to write programs that have low-level control over resources, while still ensuring high-level safety guarantees (for programs written in safe Rust). What distinguishes Rust from other systems programming languages (e.g. C/C++) is its ownership framework, which is an integral part of Rust. The ownership framework keeps track of references to different memory locations, and will free them once they go out of scope. If a program passes the compiler's checks, it meets several safety properties by construction; e.g. it has no dangling references or double-frees. Such properties must be manually ensured (or not!) in languages like C/C++. Moreover, not only does ownership remove the need for a garbage collector in the runtime environment, it frees the programmer from the responsibility to ensure certain safety-related properties (Qin et al., 2020). Due to the memory and thread safety guarantees that Rust provides, many software companies have adopted Rust in production. For example, Dropbox decided to migrate to Rust for developing their storage system, since they were not completely satisfied with the features provided by AWS's S3 or Golang (Moss, 2021).

Bugs have long been an unavoidable part of computer programming, predating Grace Hopper's literal debugging session in 1947; even today, bug-free software remains an aspirational aim for software engineering research. One step towards that aim has been studying common bug patterns (and their corresponding fixes) across multiple software systems. Research efforts categorizing common bug patterns date back to at least 1975, where Endres (1975) tried to categorize bug patterns in operating system implementations.

In this work, with the help of our tool, Ruxanne, we empirically analyze the most popular Rust projects in GitHub. Our goal is to discover common bug patterns that afflict these projects. Although we are motivated by a desire to develop automated program repair tools, we believe these insights can also be of broader service. Automated program repair (APR) (Le Goues et al., 2019; Liu et al., 2018) tools try to find bug locations in the source code (using a fault localization module) and then generate patches to fix them, possibly without human intervention, so that the modified program meets desired specifications (implicit or explicit). A powerful automated program repair system has many useful applications and could significantly simplify debugging, thus reducing software development cost (Le Goues et al., 2012). A set of common bug fix patterns can be a useful input to an APR system: it tells how to fix recurring buggy structures, thus reducing the search space for patch generation (Jeffrey et al., 2009). Yet, to the best of our knowledge, no research has sought to identify the most pervasive bug fix patterns in Rust programs.

In this work, we introduce Ruxanne, a tool for analyzing and extracting fix patterns in Rust. Ruxanne uses a novel embedding of Rust code into fixed-sized vectors. Through Ruxanne, we mined the top 18 most-starred Rust projects in GitHub to discover the most common bug fix patterns committed to their repositories. We analyzed 87,726 code changes drawn from 57,214 commits across these 18 projects. After clustering the code changes, and conducting a manual analysis, we discovered 20 groups of cross-project bug fix patterns, which we categorize as (1) general patterns and (2) borrow-checker-related patterns. We describe each of these patterns. The most common general pattern is addition or removal of struct fields, while the most common borrow-checker-related bug fix pattern is removing a call to the clone() function. Our patterns can serve as knowledge to be embedded into the design of novel Rust automated program repair tools; we present some specific potential applications of these patterns below.

If we observe undesired behaviour from a program, then there is likely a fault within the program[1]. But where is the fault? Fault localization (Wong et al., 2016) is the process of automatically finding faulty statements in a program. If a fault localization tool knows about the common bug patterns that can appear in the underlying programming language, it can prioritize candidates based on their bug-producing potential. That would result in a more robust tool—one that leverages historical data.

Moreover, knowing that certain code patterns in a programming language are more susceptible to bugs enables projects to set policies that avoid such patterns. For instance, in the context of C and C++, Cannon et al. (1991) recommend that, if at least one of the if/else sections is a compound statement, requiring braces, then both sections should have braces (i.e. they should be fully bracketed). This recommendation presumably comes from qualitative experience, as it pre-dates large-scale empirical studies. A project can choose to require that contributions follow such recommendations. Research such as ours can help formulate data-driven recommendations that reduce the number of patterns empirically linked to bugs. Monperrus espouses the claim that the fewer frequent bug patterns we keep in our software, the lower cost we can expect to pay for software maintenance (Monperrus, 2014).

Moving further afield, researchers have been exploiting deep learning techniques to create program modifier models, which can then be used for specific goals (e.g. bug fixes) (Alon et al., 2019b, 2018; Raychev et al., 2016; Bielik et al., 2016). However, most extant learning tools take fixed-size vectors as inputs. The process of compressing a variably-sized program to a fixed-sized vector is called code embedding (Chen and Monperrus, 2019). Unlike with image data, it is challenging to compress programs into vectors without any semantic information loss. Current approaches try to define the dimensions of the fixed-sized vectors in a way that accounts for all possible abstract syntax tree forms. That results in a large, sparse space for the inputs. Assuming that the deep learning model we are trying to develop is used for bug fixing, then

[1] At the extreme, a fault could be caused by moths or other hardware malfunctions.

knowledge about common bug fix patterns can help prune the dimensionality of the input space, which can help create more efficient embeddings.

Many studies have investigated detecting and categorizing bug fix patterns in general (Islam and Zibran, 2021; Madeiral et al., 2018; Pan et al., 2009), or for a specific programming language (Yang et al., 2022; Hanam et al., 2016; Campos and Maia, 2019). In this work, we study common bug patterns that appear in Rust projects. The Rust compiler's Borrow-Checker component enforces ownership rules and hence many of Rust's safety properties. Because of the Borrow-Checker's importance in Rust, we split the patterns that we present into two groups: (1) non-borrow-checker related patterns (General Patterns), and (2) borrow-checker related patterns (BC-Related Patterns).

Research Questions We formulated three research questions (RQs) which guided our empirical study about Rust bug patterns:

- **RQ1. Does our code embedding approach capture the most important aspects of bug-fixing program changes?**
- **RQ2. What are the general fix patterns in Rust, and how often do they apply?**
- **RQ3. What are borrow-checker related fix patterns in Rust, and how often do they apply?**

Contributions This paper makes the following primary contributions:

- To the best of our knowledge, this work is the first to exploit a code analysis pipeline to automatically mine Rust open source projects and extract common cross-project bug fix patterns.
- This work proposes a novel method to encode key AST information in fixed size datapoints using a semi-automatically derived weighting scheme.
- Keeping in mind the key role of the borrow checker in Rust, this work specifically identifies bug fix patterns that are related to Rust's ownership framework in addition to general bug fix patterns for Rust.
- This work finds that the most frequently observed general pattern is the addition or removal of struct fields and that the most frequently observed BC-related pattern is the removal of a clone() call.

Data Availability Statement The datasets generated and analyzed during the current study are available in the Zenodo repository, `https://zenodo.org/record/8052979`.

2 Methodology

In this section, we provide an overview of our methodology, with full details to appear in later subsections. Ruxanne uses a pipeline for finding classes of pervasive patterns in Rust programs. This pipeline parses program versions, computes differences between them, embeds them in fixed-size datapoints,

and then clusters these datapoints. After conducting a manual analysis on the obtained clusters, we refine them to obtain our proposed bug fix patterns.

We define a code change to be a modification to a program's source code that changes the program's abstract syntax tree (AST). A change pattern is a set of code changes that serves a specific purpose. Purposes include rectifying program behaviour with respect to a specific functional or non-functional requirement (bug-fixing patterns) and changes that support bug-fixes (fix-induced patterns) as well as improving code readability or maintainability (refactoring patterns). In this work, we are interested in bug-fixing and fix-induced patterns. Thus, from now on, whenever we refer to code patterns, we mean bug-fixing and fix-induced patterns.

Our approach for automatically finding pervasive code patterns aims to find clusters of similar code patterns, using existing implementations of appropriate clustering algorithms. Work in this vein generally makes the assumption that clusters correspond to classes of change patterns (Hanam et al., 2016; Campos and Maia, 2019; Yang et al., 2022). We thus want to associate code changes with datapoints; one of our contributions is a fixed-size embedding which highlights the important parts of a code change.

We implemented Ruxanne completely in Python. Our target repositories were the top 18 most-starred open source Rust projects on GitHub as of August 2021. We mined (using Pydriller) all the bug related commits and ran them through our pipeline; Section 2.1 describes how we extracted fixed-size datapoints from the code changes, and Section 2.2 describes our mining methodology in depth. Then, we applied the DBSCAN clustering algorithm (Ester et al., 1996) on the resulting datapoints. Section 2.3 describes clustering in detail, justifies our choice of DBSCAN for clustering, and explains how we tuned it to improve cluster quality.

2.1 Code Embedding

To compute the contents of a change, we need two code revisions: the revision before the change, and the revision after it. After parsing these two revisions, we will have two ASTs. Tree diff algorithms can compute the differences between two arbitrary trees. When the trees are programs' abstract syntax trees, we call their diff an ASTDiff. An ASTDiff may include an arbitrary number of semantic changes, although a best practice is to include only one semantic change in a commit. Because that best practice is not universally followed, we are interested in finding the most important change within an ASTDiff, and embedding that change in our datapoints. Section 2.1.1 describes our program parsing process, and how we obtained ASTDiff from code changes. Sections 2.1.2 and 2.1.3 provide a detailed explanation of how we select the most important semantic information out of an ASTDiff, which then allowed us to obtain clusters that contained similar datapoints.

2.1.1 Parsing Programs

Syn[2] is a Rust crate built for the Rust procedural macro implementation. However, it includes a parser which is suitable for our purposes; we simply had to write a preprocessor to transform Syn AST output into Python dictionaries. Syn handles all of Rust. Each of our datapoints specifies whether any Syn non terminal (`NT`) is present or absent, i.e. containing one dimension per non terminal. However, we are also interested in change patterns that involve the borrow checker (BC), so we add a dimension reflecting the presence of a set of BC-related elements (`BCE`) that we identified, e.g. `clone`, `Rc`, `Box`. The full list of `BCE` elements can be found in our replication package[3].

We parse two versions of a changed Rust file in a commit: the file before the commit and after it. This yields two Syn trees. Using the PLY tool[4] (a Python implementation of lex and yacc), we wrote a simple transformer from the serialized Rust AST to Python dictionaries. The change that transforms the first tree to the second one is the fix that was applied in the commit. To find this difference, we use dictdiffer[5], a Python library to find the diff of two Python dictionaries. Its output, in our context, is the ASTDiff.

2.1.2 Path Extraction

The output of dictdiffer is a list of diffs, where each diff has three parts. The first part specifies the modification type:

- 'add': A new structure has been added to the tree;
- 'remove': A structure has been dropped from the tree;
- 'change': The content of a sub structure of the first tree has changed.

The second part identifies the context of the modification, that is, the path from the root of the tree to the subtree in which the modification has occurred. The third part is the content of the change—terminals and non terminals. Ruxanne looks for changes within each root-level scope (the Rust book[6] calls these root-level scopes the *items* of a crate). That is, our assumption is that each change pattern occurs within one root-level scope. In this work, we chose to not handle patterns that involve changes in multiple functions or multiple files. However, our tool detects both continuous and non-continuous line changes within one scope.

Looking at the tree encoded in the content part of each diff, we realized that each path to a leaf represents one element contributing to the change. A Depth First Search allows us to collect all paths, and we return only the paths that we store in our datapoints (i.e. touching $\texttt{NT} \cup \texttt{BCE}$).

[2] `https://crates.io/crates/syn`
[3] `https://zenodo.org/record/7388618`
[4] `https://www.dabeaz.com/ply/`
[5] `https://dictdiffer.readthedocs.io/en/latest/`
[6] `https://doc.rust-lang.org/reference/items.html`

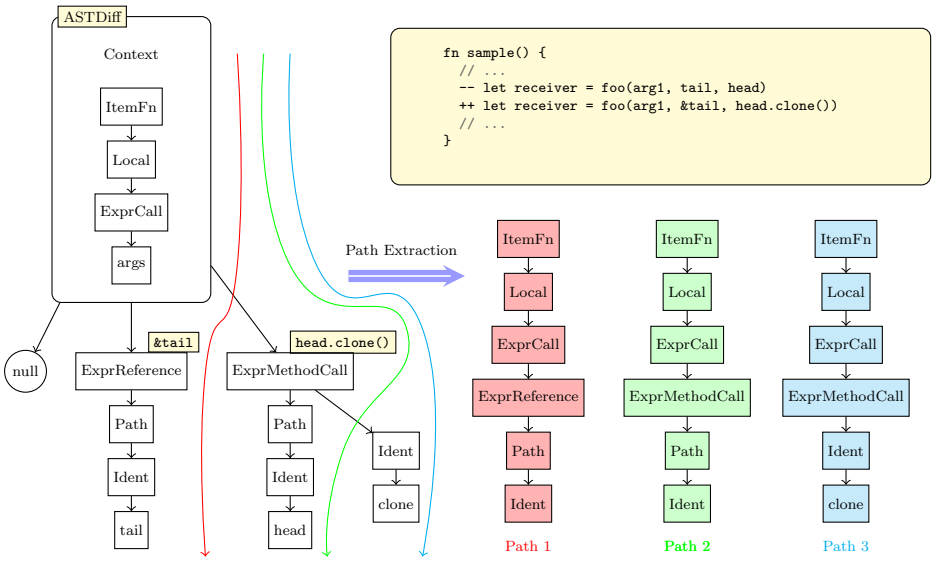

Fig. 1 The ASTDiff (left) corresponds to the code change in the yellow box (top right). A DFS of the ASTDiff finds three paths of nonterminals (Paths 1–3), shown in three different colors at right. The paths summarize (1) the addition of the ExprReference `&` before tail; (2) the change from a read of `head` to a method call (represented by ExprMethodCall), and (3) specifically a call to distinguished method `clone()`.

Running example: path extraction. Figure 1 illustrates path extraction on a simple change, starting from the ASTDiff. The change (shown in the top right corner of the Figure) replaces `tail` with `&tail` and `head` with `head.clone()`. After the change, the second argument passes a borrow of `tail` rather than sending the ownership of `tail` to the callee; and, the third argument sends a clone of `head` to the callee instead of sending `head` itself. A DFS on this tree yields three different paths, which we label "Path 1" through "Path 3" and show in red, green, and blue. A path represents a sequence of involved non terminals, e.g. the one starting with ExprReference in Path 1. It is crucial to record the order of nodes in the path, so that we can record that order in our datapoint embedding.

Note that the tree depicted in Figure 1 is not an AST; rather, it represents an ASTDiff. The paths in the ASTDiff capture the changes in the code structure between the two versions, allowing us to analyze and extract relevant information for further processing. The number of paths in the ASTDiff does not directly correspond to the number of changed elements in the source code. Instead, it is determined by the structural differences between the ASTs of the two code versions.

In our example, the change adds a `&` borrow operator to the variable `tail`, which gives rise to an ASTDiff path introducing a new parent node (`ExprReference`) for `tail`. The change also adds a call to `clone()`. This addition gives rise to two ASTDiff paths: one showing that the `head` node now

belongs to an `ExprMethodCall`, and one indicating the new sibling node for `head` representing the `clone()`. (Among the terminals, we only keep the distinguished terminal `clone` in our paths, as it is a BC-related keyword.) One path for the `&` borrow operator and two for the call to `clone()` result in three paths in the ASTDiff, even if the change only appears to include two modifications.

Code Embedding. A key question is how to transform these paths to a fixed size datapoint. We need fixed-size datapoints, as they are a requirement for well-known centroid-based and density-based clustering algorithms (Xu and Wunsch, 2005). First, we need to define a fixed set of columns. The number of columns controls the amount of information we can embed in the datapoints.

The most naive approach would be to only report the observed non terminals within the diff. That yields a fully order-insensitive representation. In such an embedding, for instance, there is no difference between two nested if statements and two if statements beside each other. On the other hand, theoretically, for full order-sensitiveness, we would need to embed all possible combinations of items from $\texttt{NT} \cup \texttt{BCE}$ as our dimensions, which yields a combinatorial explosion; this could potentially be reduced somewhat, but is still impractical. While we can implement a specific workaround to distinguish nested ifs (and we have done so), the general point about the large dimensionality of fully order-sensitive embedding still holds.

A reasonable workaround would be to pass from the full set of non terminals to a smaller set of categories (e.g. a category for larger entities like class or function definitions, and a category for smaller entities like statements and expressions), and then to record all possible combinations of these categories (similar to Hanam et al. (2016)). That ad-hoc approach reduces the number of dimensions and provides more reasonable and syntactically-correct combinations. However, it would still yield a sparse dataset.

We propose a novel representation. Similar to the naive approach, we define the columns of our dataset as the set of items $\texttt{NT} \cup \texttt{BCE}$ collected from Syn. However, to account for order-sensitivity, we introduce two additional steps. Firstly, we record the number of occurrences of items from $\texttt{NT} \cup \texttt{BCE}$ within each distinct root-level scope (e.g. structs, functions, impl blocks, etc). This captures the varying occurrence counts resulting from different element orders in the program. Secondly, we incorporate a weighting scheme, designed semi-automatically, to prioritize non terminals that we consider more salient for Rust bugs.

Running example: computing fixed-sized vectors. Figure 2 depicts the fixed-sized vectors, as described in this subsection, for the change from Figure 1. All of the columns in this vector belong to scope `ItemFn`—that is, these non terminals occur within a function. The Figure also depicts the essence of change, to be described immediately below.

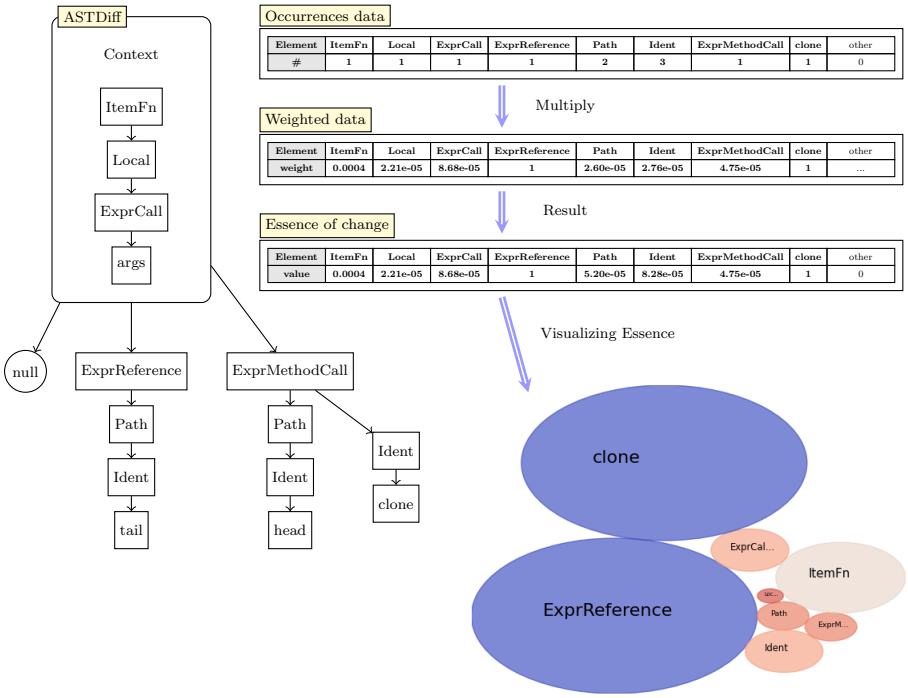

Fig. 2 Multiplying the number of occurrences of items observed in the tree by their respective weights results in the essence of change, visualized with a circle pack figure at bottom right. All non terminals here occur within scope `ItemFn`.

2.1.3 Weighting Scheme

Our main requirement is that our datapoints summarize the key changes in ASTDiffs—a datapoint should foreground the most important change in a diff. For instance, the blue path (Path 3) in Figure 1 could be described as a change in a local variable declaration; a change in a function call; calling a method of one of the function arguments; or calling clone() on one of the function arguments. The last description, in our opinion, is the most useful one, and we designed our weighting scheme to embed this value judgment. We made two observations about changes. First, we saw that items (non terminals or BC-related elements) that occur closer to AST leaves tend to be more semantically important. Second, and relatedly, items closer to the root tend to be repeated in all the paths within one ASTDiff (e.g. ItemFn, Local, ExprCall in Figure 1). We thus want to prioritize items that tend to occur closer to leaves and de-prioritize items that tend to occur closer to roots.

To design our weighting scheme, we collected empirical data about locations of item occurrences in a corpus. Specifically, we mined 20 of the most recent bug related commits of the same projects described in Table 1, and ran them through our pipeline. We recorded the total number of occurrences of

Table 1 We chose the 18 most-starred Rust projects on GitHub as target repositories for mining.

Project Name	Stars(k)	LOC	# Commits	Avg. Code Churn
denoland/deno	85.9	146055	8026	49.22
tauri-apps/tauri	52.5	50014	3041	31.67
alacritty/alacritty	42.4	30124	2036	27.53
sharkdp/bat	37.6	9496	2457	59.13
starship/starship	29.3	34888	2379	25.1
rust-lang/rustlings	30.6	5067	1466	8.92
meilisearch/meilisearch	30.2	25981	4029	279.7
sharkdp/fd	24.9	6542	1068	12.2
swc-project/swc	24.2	346501	5884	44.34
yewstack/yew	24.5	44295	2280	21.79
xi-editor/xi-editor	19.6	39292	2105	28.67
AppFlowy-IO/AppFlowy	28.4	57646	3321	29.49
firecracker-microvm/firecracker	19.6	82021	3620	58.26
nushell/nushell	21.1	163562	6088	22.38
ogham/exa	19.5	10852	1558	11.99
SergioBenitez/Rocket	18.7	68952	2135	19.75
rustdesk/rustdesk	30.2	48115	2620	72.81
tokio-rs/tokio	17.9	121400	3101	28.87

each item. Then, if $\#i$ is the number of occurrences of i, we gave i a weight of $1/\#i$. Because items that occur closer to the root also occur more often, per our observation, this gives higher weights to items that occur infrequently, i.e. closer to the leaves. Items that occur often and have lower weights are, we believe, less important in describing a change.

Furthermore, as we wanted to make sure that our tool captures patterns related to the Rust borrow checker, we manually increased the weights of the `BCE` items. The manual adjustment is why we characterize our weighting scheme as semi-automatically designed. The final weights can also be found in our replication package.

Multiplying the number of occurrences of all observed items in a change by their respective weights results in a vector of numbers. We call this vector the *essence of change*. In this vector, the value for each item shows its importance in the change.

Running example: essence of change. Figure 2 shows a complete example for computing the essence of change. The essence of change vector essentially serves as our embedding vector. The Figure shows the process for transforming the patch illustrated in Figure 1 from fixed-size datapoints (computed as described in the previous subsection) into the essence of change vector. To better illustrate the essence vector, Figure 2 also graphically presents it using a circle pack figure. The radius of each circle corresponds to the importance of the item inside it.

2.2 Mining Repositories

Now that we have set up our code analysis pipeline, we can start mining the Rust repositories. We target the top 18 most-starred Rust projects on GitHub, at the time of data collection. Table 1 summarizes our benchmarks. Because we are interested in changes, we computed (using process metrics provided by our mining library) the average code churn of the files within the projects. swc-project/swc, nushell/nushell, and denoland/deno are the projects with the most LOCs. Unsurprisingly, we captured a lot of instances from these projects in our final clusters.

We used the Pydriller (Spadini et al., 2018) library to mine software repositories. Pydriller provides APIs to extract commits from a Git repository and to search through different revisions of files. Algorithm 1 shows how we used Pydriller to run the target repositories through our code analysis pipeline and populate two databases: one for borrow-checker related patterns (which involve items in `BCE`) and one for general patterns (which don't). For each repository, we search all commits that include bug-fixing related keywords within their commit messages. Here, we chose bug-fixing related keywords based on the set introduced in Zhang et al. (2018), excluding 'nan' and 'inf'.

Next, we collect a pair of revisions for each Rust file. The pair includes the state of the file before the commit and after the commit (f_a, f_b). After parsing each revision and computing the ASTDiff, we can compute the fixed sized datapoint DP. If the datapoint contains borrow-checker related keywords, we put it in the BC-related database D_b; otherwise, we put it in the general code changes database D_g. In both cases, we augment the datapoint with the commit hash, filename, and the scope in which the change happened. For BC-related code changes, we also store the detected BC-related keywords. In summary, a datapoint is a tuple containing essence values for each element in $\texttt{NT} \cup \texttt{BCE}$ plus metadata about the change (e.g. commit hash, filename, etc.). Now our databases are ready for clustering and categorization.

2.3 Clustering Data

Having collected our datapoints, our next task is to cluster them so that we can categorize bug patterns. Like Hanam et al. (2016), we use the DBSCAN clustering algorithm for two main reasons. Firstly, in contrast to k-means, it does not require the number of clusters in advance as an input to the algorithm. Secondly, it is a density-based clustering method, which means that it detects arbitrarily shaped clusters; centroid-based methods like k-means can't detect such clusters.

DBSCAN takes two tuneable parameters. The first parameter ϵ indicates a radius distinguishing core points, border points, and outliers. The second parameter Z is the minimum number of points per cluster. We ran DBSCAN using different combinations of these two parameters. In Figure 3, we show nine different experiments, with their respective parameter values. Each subplot

Algorithm 1 Mining Algorithm

```
Input: R (target repositories)
Output: D_g (General code changes)
Output: D_b (BC-related code changes)
D_g ← φ
D_b ← φ
for r ∈ R do
    for c ∈ ExtractCommits(r) do
        if c.msg contains bug fixing related keywords then
            for {f_b, f_a} ∈ GetModifiedRustFiles(c) do
                for e ∈ ASTDiff(Parse(f_b), Parse(f_a)) do
                    DP ← GetDataPoint(e)
                    DP ← c.hash ∪ f_a.name ∪ e.scope ∪ DP
                    if IsBCRelated(e) then
                        DP ← GetBCKeyword(e) ∪ DP
                        D_b ← D_b ∪ DP
                    else
                        D_g ← D_g ∪ DP
                    end if
                end for
            end for
        end if
    end for
end for
```

specifies, for both D_g and D_b, the number of clustered points, noise points, and the number of clusters.

As a general rule, lower Z would allow more groups of nearby points to be considered to be clusters, resulting in a higher number of clusters. A large ϵ would widen the search radius, allowing more points to fall in the clusters, which results in a reduction of the number of noise points. However, such a choice might put points from different patterns inside the same cluster, resulting in ineffective clustering. Alternatively, a small ϵ would tighten the ring of search, possibly separating clusters which essentially manifest the same code changes. Also, if ϵ is really small, the clustering algorithm might not detect some clusters at all, resulting in a reduction of the number of clusters.

2.3.1 Manual analysis for parameter tuning

Since both large and small values of ϵ would steer us away from obtaining meaningful clusters, we felt the need to carry out a manual analysis of clusters for better parameter tuning. In the manual analysis, we randomly picked 50 clusters; from each cluster we randomly chose 10 datapoints. The person analyzing the cluster (the first author) then looked at the code and specified the clusters that contained more than five datapoints showing similar patterns. After conducting the manual analysis, we decided to use $\epsilon = 0.0001, Z = 5$ and $\epsilon = 0.001, Z = 5$, as our clustering parameters for D_g and D_b, resulting in 577 and 102 clusters, respectively. As a final step, we excluded clusters that contained datapoints from fewer than 3 projects. That is because we wanted

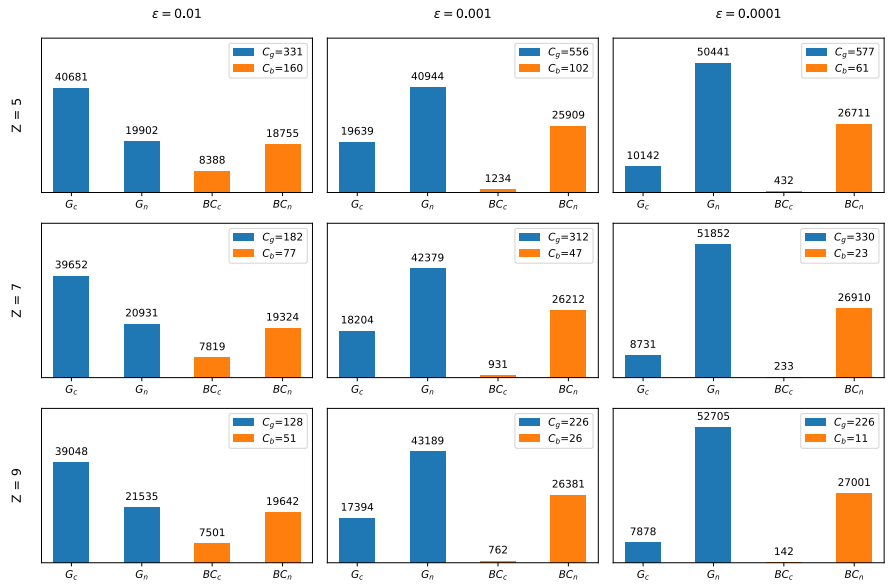

Fig. 3 Clustering results with nine different combinations of parameters ϵ and Z, respectively separated in columns and rows. In each subplot, the two blue bars in the left respectively indicate the number of clustered (G_c) and noise (G_n) datapoints (the number is shown above each bar) after applying clustering on general database D_g. The two orange bars in the right show similar data but for the borrow-checker database D_b. In each subplot's legend, C_g and C_b show the number of clusters obtained from D_g and D_b, respectively

our clusters to contain cross-project (e.g. potentially generalizable) patterns as much as possible.

2.3.2 Manual analysis for cluster selection

Like Yang et al. (2022), and following the definitions of Cotroneo et al. (2019), we divide the clusters in three different groups:

- bug-fix: The code change rectifies undesired behavior in the software system, providing fixes for a specific bug type. We also include here changes that improve program performance, as they fix the behaviour of the software.
- fix-induced: The code change is a part of a bug-fixing code and does not constitute the whole bug fix. For instance, introducing additional input parameters to a method also requires adjustments in the corresponding method invocations and does not represent the complete bug fix on its own.
- refactoring: The software behavior remains unchanged after the code change. These changes are made for improving the codebase's maintainability, readability, or encapsulation.

In this work, we were only interested in the first two groups and ignored clusters for refactoring changes. Also, we prefer reporting clusters that manifest more-specific patterns. An abstract pattern as opposed to a more-specific pattern can account for many changes in the programs. We use our judgment to prioritize patterns that we deem more specific. For instance, we deem the pattern 'Changing a clone of a variable to a borrowing of it in a function argument' more specific than 'Changing a statement in a function's body':

```
// First Pattern (more important): change clone to borrow
-- foo1(arg.clone);
++ foo1(&arg);

// Second Pattern (less important): change a statement in an fn
fn foo2() {
--  stmt1;
++  stmt2;
}
```

For our manual analysis, we analyzed all datapoints in clusters with fewer than 50 datapoints, and randomly picked 50 datapoints from clusters with more than 50 datapoints. Such large clusters account for 35/428 cross-project general patterns (out of 577 general pattern clusters) and 4/49 BC-related patterns (out of 102 BC-related pattern clusters). We exhaustively analyzed all datapoints in the 393 small cross-project general patterns and 45 BC-related clusters.

For each cluster, we read the code and the bug reports of the datapoints (if available). If we judged that the members of a cluster fit into a pattern, we wrote a natural language description of that cluster. Considering the descriptions, we linked similar clusters together as possible candidates for merging, and removed the datapoints that were not either bug-fix or fix-inducing. We reconsidered merging candidates and carried out appropriate merges. Finally, we selected 20 of the remaining clusters—the ones that manifested more-specific changes. Sections 3.2 and 3.3 present our clusters.

3 Results

In this section, we present the clusters of fix patterns that we found in our data set. RQ2 is about general patterns, while RQ3 is about borrow-checker related patterns. However, our technique fundamentally relies on our code embedding approach successfully capturing the most important aspects of program changes. We thus first evaluate the code embedding approach as our RQ1, before continuing with RQ2 and RQ3.

3.1 Evaluating the code embedding approach (RQ1)

Does our code embedding approach capture the most important aspect of program changes?

As discussed in the previous section, we used an embedding mechanism to incorporate AST information in fixed-sized datapoints which then could be used in the DBSCAN clustering algorithm. We used heuristics (i.e. a weighting scheme) while creating our datapoints to make the program elements representing the most important non terminals stand out from the complete set of non terminals seen in the AST. In addition, we also incorporated BC-related elements in our datapoints. Here we evaluate the effectiveness of our embedding.

Our evaluation aims to determine whether the most important program elements get high values in our embedding, i.e. are recognized as important. A human analyzer (paper author) inspects the actual change in the code and compares it to the visual representation of the respective data point. We used circle pack figures to visualize each data point (Collins and Stephenson, 2003), generated with the packcircles python library[7]. In these figures, the radius of the circle for each non terminal shows its importance in our embedding (Figure 2 presented an example of the circle pack figures).

We collected random samples 1000 times. In each iteration, we sampled 50 datapoints from our general and borrow-checker databases (25 each). For each database, out of the 1000 sample sets, we picked the one that minimized the sampling error (i.e. we are aiming to choose the sample that looks most like the entire dataset). The sampling error is defined as the average of the absolute differences between the sample set mean and the total mean per column (DeGroot and Schervish, 2012). Also, these random samples are drawn from the clustered datapoints. The reason behind this decision is the same as why we collected a lot of noise points: the majority of the commits involve changes in many program elements and the key changed elements are difficult to determine.

We carried out a manual experiment for evaluating our code embedding. In the experiment, the human analyzer looks at the change's source code and finds the most important element that appears in the circle pack figure, recording its rank. A value of 1 indicates that the human analyzer concurs with the algorithm, while 2 means that the human-identified most important element was ranked 2 by the algorithm. After collecting all the ranks, we then calculate Top-n Accuracy for $n = 1$ up to $n = 5$. Both authors acted as analyzer and carried out the experiment independently. To avoid inflating our final results, after comparing the ranks that the authors gave to each datapoint, we consider the larger (worse) rank in our final results. That is, if the first author gave rank 2 to datapoint D and the second author gave rank 3 to that datapoint, in the final results we gave datapoint D a rank of 3. The authors' answers were the same for 45 of the 50 datapoints.

[7] `https://github.com/mhtchan/packcircles`

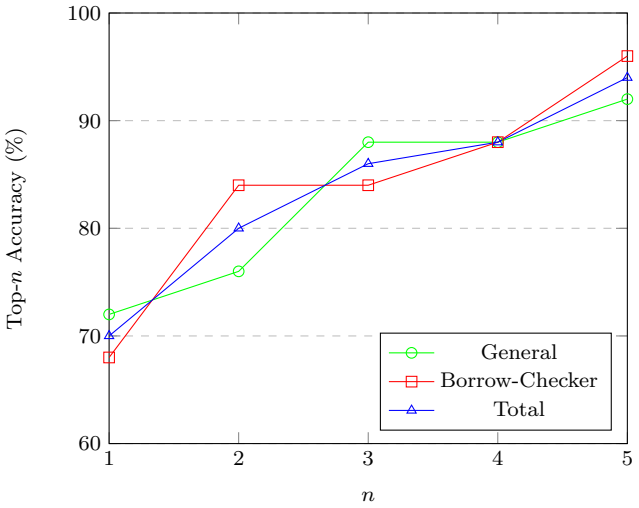

Fig. 4 Top-n Accuracy of our proposed code embedding approach. The plot shows Top-n Accuracy where $n \in \{1, 2, 3, 4, 5\}$ for both general and borrow-checker related datapoints. As shown in the plot, in more than 85% of the datapoints, the most important element is seen at least in the first three ranks, rising to about 95% for the first five ranks.

Figure 4 shows the final results. Although we judged that our results were acceptable—the result reported by our embedding was top-4 as ranked by humans 88% of the time—we investigated the other 12% as well. We found that our embedding tended to fail in change patterns where the change involves many elements (e.g. adding a whole new `impl` block and implementing new functions in it). Such failing change patterns tend to apply across projects less often, as having many changed elements make it easier for datapoints to be dissimilar.

3.2 Common Bug Fix Patterns (RQ2)

What are the general fix patterns in Rust, and how often do they apply? We collected general fixes, asked DBSCAN to create clusters, and manually analyzed them. In this subsection, we will discuss each pattern, as well as an important aspect of its value: its actionability. By actionability, we mean the possible usages of the pattern for bug repair. Other researchers can also use the empirical pattern frequency numbers that we report to build their own program repair tools, prioritizing more-frequently-repaired patterns; they are not limited to our suggested actionability. Section 4.1 contains a global discussion of our patterns and their value.

Our dataset includes 60583 datapoints for general fix patterns. We processed our dataset on a server with a Xeon Gold 5120 processor (Q3-2017, 14-core 2.2 to 3.2GHz). The data collection took around 26 hours to populate the general and BC-related databases D_g and D_b. We used DBSCAN with

Catgegory	ID	# Instances	# Projects
Attributes	G.attr.struct	81	16
	G.attr.struct.field	34	9
Struct	G.struct.field	179	17
	G.struct.field.pub	23	12
Option	G.option	22	12
Types	G.type.field	72	14
	G.type.generic	46	14
	G.type.enum.variant	39	12
	G.type.tuple	6	6
Traits	G.trait.bound	6	5
Match	G.match.pattern	41	12
	G.match.code	8	5

Table 2 Bug Fix Pattern Categorization (General Patterns)

parameters $\epsilon = 0.0001$ and $Z = 5$, resulting in 577 clusters. After manual analysis, as described in Section 2.3.2, we ended up with 12 general patterns, which we describe in Table 2. As shown in the table, we have separated the patterns based on the underlying program element (Category column). We have given each pattern a specific ID (ID column) and determined the number of instances manifesting that fix pattern (# Instances column), as well as the number of projects in which the pattern was seen (# Project column).

Since these patterns are general Rust patterns, analogous patterns can also occur in other programming languages, and we will discuss such analogous patterns when describing each pattern. We next provide a detailed description of each of these patterns.

3.2.1 Attributes

A quite common bug fix pattern in our corpus modifies attributes. In Rust, attributes enable developers to add metadata to program elements. Some of the attributes—presumably the vast majority of those involved in bug fixes—affect the semantics of the program; for instance, libraries generate different code based on the attributes that the developer specifies.

Developers can add attributes to structs, struct fields, enum variants, match expression arms, etc. The concept of attributes in Rust is similar to Python's decorators and Java's annotations. While each language may have its own syntax and specific use cases, the fundamental idea of attaching metadata to code elements is shared across these languages. Some attributes, decorators, and annotations have no run-time effect and thus could not cause bugs (if we accept a purely operational definition of "bug"). Yang et al. (2022) reported no bug patterns involving Python's decorators in their Python dataset. As we alluded to above, however, Rust has many attributes that definitely have run-time effects. We will see some of these attributes in the following examples.

G.attr.struct *Modifying the attributes of structs*

```
// G.attr.struct.add: Adding attributes to a struct
```

```
-- #[derive(Clone, Properties)]
++ #[derive(Clone, Properties, PartialEq)]
pub struct CounterProps {
    pub destroy_callback: Callback<()>,
}

// G.attr.struct.drop: Removing attributes from a struct
-- #[derive(Serialize, Deserialize, Clone, PartialEq, Eq, Debug)]
++ #[derive(Serialize, Deserialize, Clone, PartialEq, Eq)]
pub struct Subset {
    segments: Vec<Segment>,
}

// G.attr.struct.change: Changing an existing attribute of a struct
-- #[derive(Deserialize, Debug, Copy, Clone, PartialEq, Eq)]
++ #[derive(ConfigDeserialize, Debug, Copy, Clone, PartialEq, Eq)]
pub enum SearchAction {
    SearchFocusNext,
    SearchFocusPrevious,
    SearchConfirm,
    SearchCancel,
    SearchClear,
    SearchDeleteWord,
    SearchHistoryPrevious,
    SearchHistoryNext,
}
```

Description: In this pattern, the developer changes the attribute set of a struct. There are three variants of this pattern: Adding attributes to the struct (G.attr.struct.add); Removing attributes from the struct (G.attr.struct.drop); Changing the content of an existing attribute (G.attr.struct.change). These variants account for 58, 10, and 13 out of 81 datapoints in the cluster, respectively. Thus, adding attributes is the most frequent variant.

In the specific cases above, the Add change requests that additional boilerplate code be generated to evaluate partial equivalence on `CounterProps` (new functionality); the Remove change causes debug code to be removed (performance fix); and the Change change requests different deserialization code be generated. In our example, adding an attribute causes additional code to be generated for a struct and could thus, for example, fix a compile-time error in a dependency.

G.attr.struct.field *Modifying the attributes of struct fields*

```
// G.attr.struct.field.add: Adding attributes to struct fields
#[derive(Serialize)]
pub struct Retain {
    ++ #[serde(skip_serializing_if = "Option::is_none")]
```

```
    pub attributes: Option<Attributes>,
}

// G.attr.struct.field.drop: Removing attributes from struct fields
pub struct ExpandContext<'context> {
    #[get = "pub(crate)"]
    pub registry: Box<dyn SignatureRegistry>,
    -- #[get = "pub(crate)"]
    pub source: &'context Text,
    pub homedir: Option<PathBuf>,
}

// G.attr.struct.field.change: Changing attributes of struct fields
#[derive(Serialize)]
pub struct Retain {
    -- #[serde(skip_serializing_if = "Option::is_none")]
    ++ #[serde(skip_serializing_if = "is_empty")]
    pub attributes: Option<Attributes>,
}
```

Description: In this pattern, the attribute set of a struct field changes. As with ***G.attr.struct***, there are also three subtypes of this change pattern. Out of 34 datapoints in this cluster, 32 involve changes to a field attribute, while two datapoints correspond to the addition and removal of a field attribute (one for addition, one for removal). Using these attributes, the developers can control different actions they wish to apply on struct fields, in addition to providing meta-information. For instance, in the code snippet provided above, the developers chose to make Serde (a serialization/deserialization tool for Rust) call a different function (`is_empty` instead of `Option::is_none`) to determine whether to skip serializing the `attributes` field. Note that ***G.attr.struct.field*** is not a subset of ***G.attr.struct***.

Actionability of attribute change bug fixes: Because attributes have semantic effects, program repair tools can effect repairs by adjusting the set of attributes included. Given the observation that so many of our bug fixes change attributes, we propose that IDEs need to be able to propose relevant attribute changes. The example for ***G.attr.struct.change*** about changing `Deserialize` to `ConfigDeserialize` is specific to that program, but other changes would apply more generally. Also, for ***G.attr.struct.drop***, we demonstrated a case where the developer removed a `Debug` attribute. A novel type of performance program repair tool working with a run-time profiler could identify unnecessary code and remove attributes that generate it.

3.2.2 Struct

A struct enables users to define a custom type that is comprised of different types. This is a common construct across languages, and the patterns that we

describe in this section also exist in other languages.

G.struct.field *Adding/Removing a struct field*

```
// G.struct.field.add: Adding a new field to a struct
struct SnapshotService<U, R> {
    uuid_resolver_handle: R,
    ++ db_name: String,
}
// G.struct.field.drop: Removing a field from a struct
struct InnerListeners {
    pending: Mutex<Vec<Pending>>,
    -- queue_object_name: Uuid,
}
```

Description: A set of new fields are added to or removed from an existing struct. A developer adds a new field to store a new piece of data in a struct. A developer removes a field when the developer realizes that the field is not required with respect to the behaviour they want to implement (possibly because they have moved it elsewhere).

G.struct.field.pub *Modifying the access modifiers of struct fields*

```
// G.struct.field.pub.add: Adding pub to a field
pub struct HtmlBlock {
    -- content: BlockContent,
    ++ pub content: BlockContent,
    brace: token::Brace,
}

// G.struct.field.pub.drop: Removing pub from a field
pub struct Languages {
--  pub named: HashMap<LanguageId, Arc<LanguageDefinition>>,
++  named: HashMap<LanguageId, Arc<LanguageDefinition>>,
    extensions: HashMap<String, Arc<LanguageDefinition>>,
}
```

Description: Adding `pub` to a struct field makes it possible to access that field from external modules. In the instances of this cluster, adding the `pub` access modifier happened in bug fixing changes where the developer needed to access a field which had not been marked for public access. On the other hand, public access might be revoked if the developer realizes there are no external accesses to the field, and that such accesses are not desirable, e.g. for encapsulation purposes.

Actionability of struct changes: The patterns related to access modifiers suggest an opportunity for repair tools to make sure that the accessibility of program entities follows the user's specifications, enhancing encapsulation,

security, and maintainability. Additionally, the empirical numbers revealing the frequency of struct field modifications in Rust shed light on the significance of these bug fix patterns and emphasize the need for further exploration on how program repair tools can effectively modify code based on changes in struct fields to repair program behavior.

3.2.3 Option

G.option *Changing field type T to T⟨Option⟩*

```
#[ast_node("MediaRule")]
pub struct MediaRule {
    pub span: Span,
    -- pub media: MediaQueryList,
    ++ pub media: Option<MediaQueryList>,
    pub rules: Vec<Rule>,
}
```

Description: Rust is a null-safe language, meaning that object references cannot take on null values. If a developer wants a variable to contain the equivalent of null, they must use the `Option` type. `Option` is essentially an `enum` with two variants: `Some`, which indicates that a variable has a value; and `None`, which indicates the absence of any value. In this change pattern, the developer changes the type of a struct field to an `Option` of that type, hence enabling them to store no value in that field. For instance, in the Rust project swc-project/swc (a fast TypeScript/JavaScript compiler), the developer needed to modify the CSS parser source code to account for empty @media queries[8]. As shown in the snippet above, they changed type `MediaQueryList` to `Option<MediaQueryList>`, and modified the other parts of the source code accordingly.

Actionability of adding Option to a type: This pattern occurs as a *fix-induced* change—a repair tool might be carrying out a broader change and need to make smaller changes like this one in the process. Our empirical results suggest that this type of change does occur in practice and will need to be part of a program repair tool's arsenal.

3.2.4 Types

Rust is a statically-typed language, and like in other statically-typed languages, some change patterns are associated with changes in types used in different program structures. Here, we introduce four bug fix patterns that are associated with types in Rust:

G.type.field *Changing a struct field type*

[8] https://github.com/swc-project/swc/commit/75a14f98b7370226115ee24eec6eb8c802bd4837

```
pub struct Manifest {
  -- pub substitutions: HashMap<String, &'a str>,
  ++ pub substitutions: IndexMap<String, &'a str>,
}
```

Description: The type of a struct field changes. This is a common change pattern in all statically-typed languages and can happen for various reasons. For instance, the developer may be implementing a new feature or fixing the behaviour of the program. Also, it can happen for refactoring purposes or for performance enhancement.

For instance, in the Rust project starship/starship (a cross-shell prompt), to preserve the insertion order of the `substitutions` field, the developer changes its type from `HashMap` to `IndexMap`[9].

G.type.generic *Changing a generic type parameter*

```
#[derive(Debug, Clone, PartialEq)]
pub struct Anchor {
    -- point: Point<usize>,
    ++ point: Point<isize>,
    side: Side,
}
```

Description: This is a change of the type parameter of a struct field. This change pattern can occur for the same reasons as for ***G.type.field***.

G.type.enum.variant *Change in enum variant value type*

```
#[derive(Debug, Clone, PartialEq, Eq)]
pub enum ResolvedDependency {
  Resolved(ModuleSpecifier),
  -- Err(String),
  ++ Err(ResolvedDependencyErr),
}
```

Description: In Rust, developers can specify the type of the value that an enum variant can store. Using this feature, they can avoid having to use a struct along with enum variants. This change pattern is associated with a change in enum variant types.

G.type.tuple *Changing the type inside a tuple*

```
#[allow(dead_code)]
pub struct CoreState {
    // Other fields
    -- pending_views: Vec<ViewId>,
    ++ pending_views: Vec<(ViewId, Table)>,
```

[9] https://github.com/starship/starship/commit/4de9e43cff46c834bd340d24a02fc95d85310a33

```
    peer: Client
}
```

Description: In this pattern, an extra type is added to an existing type, making it a tuple of multiple types. Alternatively, the change might be dropping types from the tuple and making it a smaller-arity tuple.

3.2.5 Traits

G.trait.bound *TraitBound change*

```
-- pub struct EventLoop<T: tty::EventedReadWrite> {
++ pub struct EventLoop<T: tty::EventedPty> {
    poll: mio::Poll,
    // Other fields
}
```

Description: The trait bounds are the functionalities that we require from our parametric types. This concept more-or-less corresponds to interfaces in other languages, although the mapping is not exact (e.g. it is possible to implement traits for others' types). This change pattern relates to modifications in trait bounds, which can happen as part of bug fixes or for refactoring.

3.2.6 Match

In Rust, the match control flow construct is used for comparing a value against multiple patterns, and executing the code associated with the first pattern that matches. A pattern and its associated code are called a match arm. Match statements are similar to switch statements in other languages, like C, although Rust match statements are able to match more complicated patterns.

G.match.pattern *Change in a match arm's pattern*

```
-- Value::Primitive(p) => {
++ UntaggedValue::Primitive(p) => {
    let _ = builder.add_empty_child(p.format(None));
}
```

Description: In this pattern, a change occurs in an arm's pattern. Like the code snippet provided above, it can be a change in a match pattern's type. It also can be matching to a different enum variant or string. The fix pattern can happen both for refactoring or bug-fixing purposes.

G.match.code *Change in a match arm's code*

```
Operation::Insert(insert) => {
    -- inverted.delete(insert.count_of_utf16_code_units());
    ++ inverted.delete(insert.utf16_size());
}
```

Catgegory	ID	# Instances	# Projects
Clone	BC.clone.drop	83	10
	BC.clone.ref	4	3
Ref and Deref	BC.ref.add	17	7
	BC.deref.to_string.add	4	3
Mut	BC.mut.add	11	6
	BC.mut.drop	42	15
Vector	BC.vec.slice	8	6
Lifetime	BC.lifetime.static	8	6

Table 3 Bug Fix Pattern Categorization (BC-Related Patterns)

Description: The block of code associated with a pattern in match arm can contain multiple statements. In this fix pattern, a change happens in at least one of these statements. Similar to the previous fix pattern, this also can be due to refactoring or debugging purposes.

3.3 BC-Related Bug Fix Patterns (RQ3)

What are borrow-checker related fix patterns in Rust, and how often do they apply?

We collected 27143 datapoints for borrow-checker related fix patterns. We then applied DBSCAN with parameters $\epsilon = 0.001$ and $Z = 5$, resulting in 102 clusters. After manual analysis (Section 2.3.2), we ended up with 8 borrow-checker related fix patterns. Table 3 describes BC-related patterns, analogously to Table 2 for general patterns. Since the Borrow-Checker is unique to Rust, patterns in this section generally do not have direct counterparts in other languages. We next provide a detailed description of each of these patterns.

3.3.1 Clone

BC.clone.drop *Dropping clone*

```
start_plugin_process(
--  manifest.clone(),
++  manifest,
    self.next_plugin_id(),
    self.self_ref.as_ref().unwrap().clone(),
);
```

Description: This pattern is referred to as redundant clone in Clippy Lints. Clippy[10] is a linter tool for Rust that catches common mistakes. There are more than 500 lints included in Clippy. Since detecting whether a clone is redundant is undecidable, Clippy conservatively estimates redundancy. By definition, removing a clone that is indeed redundant does not modify functional aspects of

[10] https://github.com/rust-lang/rust-clippy

the program behaviour. (Removing a non-redundant clone introduces a compile error or, worse yet, a bug.) However, a removal of a redundant clone can greatly improve the nonfunctional property of performance, if the program was formerly cloning large objects.

BC.clone.ref *Dropping clone and adding borrowing*

```
-- let field_id = schema.get_or_create(attribute.clone())?;
++ let field_id = schema.get_or_create(&attribute)?;
```

Description: Unlike the previous pattern, this pattern is not detected by Clippy. The change happens when a developer realizes that the cloning of a variable is unnecessary, but also wants to keep the ownership of the variable within the current scope (rather than passing it to a callee). That is why, in this pattern, a clone is turned into a borrow.

Like the previous pattern, this change does not affect the program behaviour; it is done for performance purposes. In one of the commits[11] of the project tauri-apps/tauri, changing a clone to borrowing significantly improved CPU usage: in the commit history, developers discussed how the cloning of types such as HashMap was expensive.

Actionability of clone removal: Like Clippy, a program repair tool can implement clone removal by reasoning about whether the clone operation was necessary. If the clone is unnecessary, the tool can replace it with a reference (as seen above), thus reducing the overhead of unnecessary copying.

3.3.2 Ref and Deref

BC.ref.add *Adding Borrowing*

```
let repo = replace(&mut *contents, Processing).inner_repo();
-- let statuses = repo_to_statuses(repo, &self.workdir);
++ let statuses = repo_to_statuses(&repo, &self.workdir);
```

Description: This change pattern happens when the developer now needs to take back ownership of an object that was previously passed to a callee (or other scope) and never returned. Borrowing allows the first scope to once again act on the object.

As an example, the commit in starship/starship excerpted above[12] shows that the developer now borrows the value of `repo_dir` to keep the ownership of the object in the current scope. This object can then be used in a subsequent function invocation (`remove_dir_all(repo_dir)`), which fixes a bug.

Actionability of adding borrowing: This pattern often occurs as part of a larger change, making it challenging to create a generalized repair tool that

[11] https://github.com/tauri-apps/tauri/commit/a280ee90af0749ce18d6d0b00939b06473717bc9

[12] https://github.com/starship/starship/commit/56d475578ea508631275772127f49a6949fea6b0

covers all possible scenarios. Typically, when a reference is passed instead of transferring ownership, the variable is used again after returning from the call. Although it is not obvious how to create a comprehensive repair tool for such cases, empirical results underscore the significance and prevalence of this fix, warranting further investigation.

BC.deref.to_string().add *Adding a dereference before calling to_string()*

```
for p in wixobjs {
--  args.push(p.to_string());
++  args.push((*p).to_string());
}
```

Description: This is a change that in all instances (in our benchmarks) has been proposed by Clippy—commit messages indicated the use of Clippy for this fix. The change applies on an object prior to calling `to_string()` on that object. In this change, a dereference operator is added to that object. The reason behind the change is that the object is a reference type of `T`, where `T` directly implements `to_string()`. Adding the dereference makes the compiler instead use the specialized implementation of `to_string()` and not go through slower string formatting methods[13]. This change is aimed at improving performance; the functionality remains unchanged.

3.3.3 Mut

BC.mut.add *Adding mutability*

```
parser::Parser::new(args)
        .parse_module()
--      .map_err(|e| {
++      .map_err(|mut e| {
            e.emit();
            ()
        })?
```

Description: This change happens when the developer needs to mutate a variable which was previously immutable in a scope—they thus need to add the `mut` keyword. We never observed this pattern in refactoring-only changes; it always accompanied a change in program behaviour.

Actionability of adding mutability: A repair tool can use this pattern to introduce mutability to the target variable and modify other parts of the program accordingly. For instance, if the target variable has associated `impl` blocks, the change may involve modifying method annotations to ensure that the calling object is mutable, aligning with the desired behaviour.

[13] `https://rust-lang.github.io/rust-clippy/master/index.html#inefficient_to_string`

BC.mut.drop *Dropping mutability*

```
-- let mut tx = tx.clone();
++ let tx = tx.clone();
```

Description: The developer drops the mut keyword before a variable, as they do not mutate the variable (which is statically checkable). This redundant mutability is reported by Clippy and can also be removed by it. Such a change may help future-proof the code against unintended future changes: mutability must be explicitly added back before the variable can be mutated.

3.3.4 Vector

BC.vec.slice *Changing a Vec reference to Slice*

```
pub fn expand_delimited_square(
--  children: &Vec<TokenNode>,
++  children: &[TokenNode],
) -> Result<hir::Expression, ParseError> {
    // body
}
```

Description: In a struct field definition, or in the types of a function's formal parameters, type `&Vec<T>` changes to slice `&[T]`. Clippy can detect opportunities to apply this pattern. This simplifies the code; slice types `&[T]` or `&str` are sufficient for most use cases.

3.3.5 Lifetime

BC.lifetime.static.drop *Dropping static lifetime*

```
-- const QUEUE_SIZES: &'static [u16] = &[QUEUE_SIZE];
++ const QUEUE_SIZES: &[u16] = &[QUEUE_SIZE];
```

Description: This change pattern removes the static keyword from a const variable declaration. This change can be applied using Clippy. A static lifetime associated with a variable indicates that the variable lives for the entire lifetime of the running program, and is not bound to a specific scope. If the presence of static is not required, it is better that to omit it, as keeping it might create complicated types in the program.

4 Discussion

In this work, we used Ruxanne to successfully mine 20 cross-project bug fix patterns. We presented these patterns in two groups: 12 general patterns and 8 BC-related patterns. Clippy is able to detect 5 of the 8 BC-related patterns. Fundamentally, Clippy cannot detect some of the remaining patterns. For instance, we proposed pattern *Adding mutability*. In the Rust context, this

change anticipates a future change to the code which will perform a mutation of a previously immutable value. Predicting the future is beyond Clippy's remit. To our knowledge, the remaining patterns (3 BC-related patterns and 12 general patterns) cannot be reported by current linting tools.

In each group, we categorized the bug fix patterns based on their underlying program element. The patterns encompass a wide variety of program elements both in the general and BC-related groups. This supports a conclusion that our weighting scheme is not biased towards specific program elements. Also, our patterns are all cross-project fix patterns: each reported fix pattern has been seen at least in three different projects.

While we introduce patterns that reflect changes at different locations in the code, such as *BC.ref.add*, which by borrowing a variable instead of taking ownership enables users to use the variable in subsequent locations, we acknowledge that it is possible that a large patch may be harder to cluster, because it has to resemble other patches. Additionally, we believe that any clustering-based approach will likely place larger diffs further apart for the same reason that ours does: there is just more going on. Nevertheless, there is nothing inherently penalizing a larger diff. Also, we point out that the related work by Pan et al. (2009) specifically excludes diffs with seven statements or more, while we allow larger diffs.

All of our parsing, path extraction, weighting scheme, and clustering modules, along with our final results, can be found in our replication package, which we provide for verification, reusability and further extension. Weightings can be modified to make the embedding focus on a specific set of elements for building different code embeddings.

4.1 Patterns Present and Missing

When presenting each pattern, we discussed actionability for the most important actions. Here, we discuss the implications of our most important patterns. *In summary, program repair tools must understand semantics of attributes and should be performance-aware.* It may also be useful for tools to be able to automatically infer immutability. Here, we discuss the top ten most popular patterns (8 general patterns and 2 BC-related patterns).

The most common general pattern and the most popular pattern overall is modifying struct fields with 179 datapoints. Unfortunately, struct field modifications are neither interesting nor actionable; they are simply used to enable other changes.

The most common borrow-checker-related pattern and second-most-common pattern overall is dropping `clone()`. This pattern is performance-related; when correctly applied, it has no semantic impact on the program, but reduces resource consumption. Rust is used in situations where performance is important, and the number of instances of this pattern confirms that developers do work on performance issues. This implies that *program repair tools would*

benefit from being performance-aware, rather than limiting themselves strictly to functional properties of the code.

Attribute modifications are interesting changes that account for the third- and ninth-most popular patterns. From a language design perspective, the fact that there are so many attribute changes marked as bug fixes illustrates the semantic importance of attributes in Rust, and *implies that program repair must understand the effects of attributes*. Yet, different attributes have different, and often library-dependent, semantic meanings. Proposing specific repair techniques that reason about attributes is beyond the scope of this work, but our results suggest that they are important to develop.

The fourth- and fifth-most popular patterns, changing the type of a struct field or of a generic type parameter, are difficult to make general statements about; there are many unrelated idiosyncratic changes in these clusters. Such changes look the same, but have different purposes. One example we showed changed a `HashMap` to an `IndexMap`, preserving iteration order. That specific nondeterminism bug could be automatically repaired upon direction from the developer. Another change modified `usize` to `isize`, perhaps reflecting a requirements change to allow signed integers. IDEs could support refactorings that modify types. We find it difficult to propose further general work in this direction.

The sixth-most-popular pattern is the Rust-specific change of dropping mutability (i.e. making data immutable). Tools like ReImInfer by Huang et al. (2012) (for a Java extension), and techniques like the one proposed in Eyolfson (2018) for C++, aim to infer that data is immutable. In the Rust context, it is easier to make data immutable: the semantics for immutability are clear. Immutability is part of the core Rust language. Immutability escape hatches in C++ complicate issues, but in Rust, the escape hatch is the clearly-marked `unsafe` block. This pattern thus suggests that *tools for inferring Rust immutability could be viable*.

We are unable to say anything meaningful about the seventh- and eighth-most popular patterns, `G.match.pattern` and `G.type.enum.variant`. They do not seem particularly helpful for automatic program repair.

The tenth-most-popular pattern, making a struct field `pub` and hence visible to external modules, suggests that tools to help developers manage visibility would be helpful. Public accessibility of methods and fields very much affects library compatibility and usability.

Conversely, we were surprised to find that Ruxanne reported no patterns associated with certain Rust-related features. For instance, we did not observe any patterns associated with the Rust `move` operator, which has been known to plague many beginning Rust programmers. `move` is used to transfer the ownership of values to a closure. In some cases (e.g. multithreading), not moving the ownership could create invalid references. Rust compiler checks prevent such errors. Our speculation is that, since Rust enforces strict compiler checks, programmers usually get many compile-time errors, but they only push their code once it is free of compile errors. We would not observe patterns which do not make it to the repository.

To compare the importance of bugs versus compilation errors: we consider bugs as potentially more critical. This is because a bug, if left unresolved, can remain dormant within the system and then inadvertently get deployed, whereas a program with compilation errors cannot be shipped by developers.

We summarize some learnings from our bug patterns:

- Rust program repair tools must understand the effects of attributes.
- Program repair tools should be performance-aware.
- Tools for inferring Rust immutability could be viable.

4.2 Usefulness of Rust Bug Patterns

Having discussed the most frequent patterns and their implications on repair tools, we continue by discussing potential features for repair and refactoring tools in more detail. We believe our results can help with the development of program repair and refactoring tools for Rust. We have discussed the actionability of some of the general patterns and three BC-related patterns that are not detected by Clippy.

For instance, as we have discussed, for the common pattern of *Modifying the attributes of structs*, a code linter tool could search for correct attribute lists within the codebase (Forrest et al., 2009), leveraging the assumption that proper attribute usage can be found elsewhere in the code. Also, the linter can analyze the usages of the struct to determine the necessity of adding the attribute. Assuming that attributes cause additional code to be generated, removing unnecessary attributes not only optimizes the codebase but also eliminates unnecessary overhead and reduces binary size.

Similarly, we presented the pattern *Dropping clone and adding borrowing*. As discussed, Clippy does not detect this pattern, and repetitive clones can be computationally expensive. Using state-of-the-art methods for program repair, one could design a tool to recognize this pattern and change variable cloning to simple borrowing.

We believe that our insights can greatly aid researchers in creating effective IDE tools tailored for Rust development. Moreover, the empirical frequency numbers associated with each pattern, highlighting its prevalence, can provide valuable guidance for the development of program repair tools.

4.3 Threats to Validity

There are two main threats to the internal validity of our work: (1) A threat to internal validity is confounding, where changes to what shows up in the embedding are not due to changes in the code being embedded. In this case, the weighting scheme may contribute to confounding because we manually adjusted the weights. Nevertheless, our weighting scheme can be readjusted

to find different patterns, which is why we made our pipeline publicly available. (2) Our code embedding approach is based on the frequency of observed program elements in ASTs, and we use DBSCAN as our clustering algorithm. Other code embedding methods and clustering algorithms (e.g. SLINK) might output new clusters that our pipeline is unable to find.

We discuss three threats to external validity: selection bias, incorrect commit/bug information reported by developers, and changes to the Rust programming language. (1) Selection bias is an issue because our benchmarks may differ systematically from the set of Rust programs in the world. Our selection of projects might be biased and therefore we might not have presented patterns that may exist in other projects. We aimed to mitigate this threat by choosing all of the most-starred projects, but this biases towards popular projects by definition. (2) Another threat to external validity can be the developers reporting commits as bug fixing commits while they are not really fixing functionality. Similarly, a commit message might not contain our target keywords while the commit is associated with bug fixing changes. This is simply an underlying assumption of our work and may cause us to miss some bugs; we do not believe it should be a systematic threat, and is shared by much other work on understanding bug fixes. (3) A third threat is that Rust is a relatively new programming language, and its syntax is prone to evolving over time. Changes in Rust's grammar, such as the introduction of new keywords like `async/await` in version 1.39, have the potential to introduce new bug categories or modify existing ones. Thus, one must consider language evolution as a potential external threat to the validity of our research findings; this threat can be mitigated by redoing our analysis in the future using our methodology with updated versions of our published artifacts.

5 Related Work

Automated program repair (APR) aims to debug faulty source code without human involvement, sometimes using test cases to guide the repair. Before a repair tool modifies code, it will typically use a fault localization module to find fault locations. The fault localization modules that we are aware of rank program statements based on suspiciousness; one example is Tarantula by Jones and Harrold (2005). Often, suspiciousness is calculated using a statistical model which relies on the observation that buggy statements are executed mostly by failed test cases (Naish et al., 2009; Xie et al., 2013).

5.1 Automated Program Repair

We next discuss a number of approaches for automated program repair. First, however, we discuss previous work that aimed to classify bugs—program repair must have a model of what it is attempting to repair in the first place. We then present two mindsets for program repair: search-based and pattern-based.

Finally, we survey related work in automated patch correctness assessment, which aims to ensure that changes by automated repair tools are valid.

Bug Classification. A foundational bug study dates back to Endres (1975), which classified the bugs that were fixed in a particular release of IBM's DOS/VS operating system, written in the DOS Macro Assembler language. Endres contended that categorizing bugs can aid in uncovering error causes, fixing them, and preventing their recurrence. The Group B of bugs in that work include, among others, bugs that could be prevented by the use of a different programming language, e.g. "assignment, loading, or saving of address registers forgotten," which is somewhat similar to our Rust-specific bug patterns. Later on, Knuth (1989) analyzed the errors that he made in the development of TEX, written in the WEB literate programming language. This was perhaps one of the first works in this domain where there is a publicly-available artifact (the TEX source code) supporting the data (i.e. the bugs identified), unlike the proprietary DOS/VS code. Knuth includes 15 categories of errors; the most relevant ones to this work are "language liabilities," "mismatches between modules," "reinforcements of robustness," and "surprising scenarios." Flanagan and Felleisen (1998) described MrSpidey, an interactive static debugger for Scheme (a dialect of Lisp). MrSpidey is a front-end to an ESC-style static verification engine which aims to detect bug patterns where operators are passed invalid arguments. The engine relies on program invariants and presents information about attempted proofs of the invariants to the developer. Most recently, Pan et al. (2009) analyzed seven large-scale widely used Java projects and manually extracted 27 common bug fix patterns.

Moving from bug classification to our application, program repair, we follow the taxonomy of Liu et al. (2018) and propose two mindsets: search-based program repair and pattern-based program repair. Our work applies to both mindsets, but is most directly applicable to pattern-based program repair.

Search-Based Program Repair. One line of research in program repair follows the "competent programmer" hypothesis (Gopinath et al., 2015): syntactically, a faulty program is not that far away from its correct version. This hypothesis suggests the search-based program repair mindset. If the hypothesis holds, we can develop mutation operators (which change an expression or a statement) and apply them to a list of fault locations provided by the fault localization module. So, we loop through possible mutations; if the mutation of a statement does not cause all expected-successful test cases to succeed, we assume that the statement is correct and move on to the next fault location. Having a set of bug-producing patterns, like the ones we are proposing, will help researchers design targetted mutation operators that leverage domain specific insights.

Use of mutation operators in program repair became popular with GenProg (Forrest et al. (2009); Nguyen et al. (2009)), a program repair tool that uses genetic programming. The main idea behind this tool is that statements follow certain patterns in a codebase. Therefore, if we find the correct version of a faulty statement somewhere in the program, we can use that version in

place of the faulty original. This tool showed promising results as it managed to find patches without any additional annotations or human involvements.

However, Arcuri and Briand (2011) observed that GenProg mostly found patches while carrying out random initialization—that is, before GenProg's evolution begins. Their tools TrpAutoRepair (Qi et al., 2013) and RSRepair (Qi et al., 2014) advocate that test case prioritization is necessary while using genetic programming, as fitness evaluation is an expensive part of the repair pipeline.

Tan and Roychoudhury (2015) used mutation repairing for fixing regression bugs. They categorized common code changes in real-world regressions after studying 73 program evolution benchmarks; our approach also yields a set of common code changes, for Rust, using a different set of changes. Tan and Rovehoudhury used their categorization to design mutation operators, and drew from those operators to repair faulty program locations.

Pattern-based Program Repair and Mining Bug Fix Patterns. The main intuition in pattern-based program repair methods is that bug fixes follow certain patterns. Thus, having a collection of common bug fix patterns is a crucial step for developing such repair tools. For this mindset, the search is for a program fix (i.e. a delta) rather than for a correct program.

Pan et al. (2009) used a bug fix pattern extractor tool to automatically parse and detect bug patterns within bug fix hunks with fewer than seven statements. They ignored larger bug fix hunks—they claimed that such hunks tended to exhibit random changes rather than having patterns that they could derive meaning from. Though they analyzed Java projects, their reported patterns were not specific to Java, in the same way that our general patterns are not specific to Rust. Martinez and Monperrus (2015, 2012) exploited the frequency of observed patterns to introduce a heuristic patch searching method. In their empirical evaluation, they concluded that choosing repair actions probabilistically (weighted by frequency) can help with reducing the search space, hence creating a more effective repair tool.

Hanam et al. (2016) conducted similar research, but for finding pervasive bug fix patterns in JavaScript. They performed a large-scale study of bug fix patterns by mining 105K commits from 134 server-side JavaScript projects. Like us, they used the DBSCAN clustering algorithm and divided bug fixing change types into 219 clusters, from which they extracted 13 pervasive cross-project bug fix patterns.

Yang et al. (2022) proposed a mining approach to detect Python bug fix patterns by studying fine-grained fixing code changes. They also examined how many bugs could be fixed using automated bug fixing approaches. Moreover, they evaluated the fix patterns that they detected and concluded that 37% of the buggy codes could be matched by the fix patterns they had found.

We would characterize our work as the Rust version of what was done in Hanam et al. (2016) and Yang et al. (2022) for JavaScript and Python, respectively, but with additional infrastructure to create a representation of the Rust code using fixed-size vectors. Like Hanam et al. (2016), we used

DBSCAN to cluster fixes. Moreover, similar to Yang et al. (2022), we introduced patterns in two different categories. In our case, there were general and language-specific patterns. The Rust language-specific patterns were related to the borrow checker (BC patterns). However, unlike our paper, Yang et al. (2022) needed to do a much more elaborate manual analysis, as they had only collected general information about the single hunk changes (e.g. the number of variables, or arguments).

Automated Patch Correctness Assessment. The final step in automated program repair is to validate the proposed changes. One substantial challenge is overfitting, where synthesized patches may pass test cases but fail to be correct. Automated patch correctness assessment (APCA) aims to inspect the correctness of generated patches using dynamic and static analyses. While our project has a different objective than APCA, we share similarities in design decisions with APCA systems, particularly in feature extraction. Ye et al. (2021) propose ODS, an overfitting patch detection system, which assumes that code features capturing universal correctness properties can be utilized to classify overfitting patches across program repair systems and software projects. ODS extracts 202 manually crafted code features by comparing fixed and buggy code, and employs supervised learning to build a distributional model for classifying unseen patches, achieving over 70% accuracy in evaluations on Defects4J, Bugs.jar, and Bears suites. Similarly, our feature extraction involves manual crafting of features. In another study, Tian et al. (2022) present an enhanced APCA model that transforms the problem into a question-answering scenario and incorporates a neural architecture to learn semantic correlations between bug reports and commit messages. Additionally, Lin et al. (2022) introduce CACHE, a neural-based context-aware AST embedding method that captures both the changed part and the unchanged part (context code) of patches. Evaluation results demonstrate that context knowledge significantly enhances the performance and accuracy of systems that attempt to automatically assess patch correctness (which is not our goal).

5.2 Code Embedding

Recent advances in deep learning have motivated researchers to build models for various types of data. Machine learning models require a numerical representation of the data to feed into the model. To realize this mapping for natural language text, researchers have proposed *word embedding* to embed word information in fixed size vectors. As programming languages are not that different from natural languages (Hindle et al., 2016), similar mappings have been suggested for computer programs (Chen and Monperrus, 2019).

Alon et al. (2019b) presented code2vec, a neural model for representing code snippets as fixed size vectors. In their approach, they extracted flattened paths from the AST, and stored all the leaf-to-leaf paths. The intuition behind

this decision was that leaf-to-leaf paths tend to encode more semantic information than root-to-leaf paths. They associated a fixed size vector with each path, and fed these as inputs to a neural network that learns how to aggregate all these paths to a single embedding. Our code embedding approach is simpler than code2vec's, as we have a specific (and simpler) goal—we aim to cluster bug patterns, not predict method names. Moreover, unlike us, code2vec does not offer a general code embedding method, since the final embedding depends on the output layer of the neural network. That is, if we change the goal of method name prediction, the embeddings would be completely different. Also, unlike many embeddings, our code embedding is interpretable: our columns are associated with non terminals. We have used the interpretability in our experiments. In code2vec, the layout of the final vector is a parameter of the system and is not related to the underlying programming language. Building on the same ideas, as a subsequent work, the same authors also introduced code2seq (Alon et al., 2019a), an approach for transforming a code snippet to a sequence encoding. They evaluated their work on three seq2seq use cases: (1) method name prediction, (2) code captioning, and (3) code documentation. code2seq proved to have a better understanding of syntactical structure of the code and outperformed previous neural machine translation systems.

Hoang et al. (2020) introduced CC2Vec, a neural network model that uses log messages accompanying code changes to learn representations capturing semantic information. By addressing challenges in distinguishing between added and removed code and incorporating an attention mechanism, CC2Vec outperforms previous techniques for code change representation. The model employs a hierarchical attention network, which incorporates a multi-level bidirectional GRU recurrent neural network, to encode information about changed code tokens, lines, and hunks into vectors representing added and removed changes. These vectors are then combined and fed into a hidden layer for predicting the target function. Evaluation across log message generation, bug fixing patch identification, and just-in-time defect prediction tasks demonstrates the superior performance of CC2Vec compared to the previous state-of-the-art approach in all three tasks.

Our work is more like that of Hanam et al. (2016). However, our novel program embedding approach differs from theirs on certain key decisions. Their work introduced the Feature Properties table, which is a categorization for program elements. Embedding programs based on this table resulted in an order-sensitive embedding. However, it made the datapoints sparse; our datapoints are more dense than theirs. Specifically, we have over 11× fewer dimensions than they do.

5.3 Refactoring tools for Rust

Clippy is widely known as the best code refactoring tool for Rust. It can identify over 600 common mistakes and offer automated fixes. As discussed in Section 3.3, we found that Clippy can detect and address five of our borrow-

checker-related patterns. Beyond Clippy, Sam et al. (2017) introduced a refactoring tool that uses the Rust compiler infrastructure to perform various refactoring operations, including renaming variables, arguments, fields, functions, methods, structs, and enumerations, as well as inlining local variables, and reifying and eliding lifetime parameters. In a more advanced use case, Ling et al. (2022) proposed CRustS, a tool designed for refactoring C2Rust output. C2Rust is a C to Rust transpiler that converts C code into Rust syntax but retains the unsafe semantics of C. By relaxing the constraint of preserving semantics during transformations, CRustS increases the proportion of transformed code that successfully passes the safety checks of the Rust compiler.

5.4 Code Patterns in Rust

To our knowledge, our work is the first to use an automated mining and code analysis pipeline to find pervasive patterns in Rust open source projects, even if previous studies have (manually) investigated common bug patterns in Rust.

Qin et al. (2020) conducted the first empirical study on real-world Rust program behaviours. They manually inspected 850 unsafe code usages and 17 bugs across five open-source Rust projects, five Rust libraries, two online security databases, and the Rust standard library. They analyzed the motivation behind unsafe code usage and removal, in addition to recording 70 memory-safety issues and 100 concurrency bugs. They also provided Rust programmers with some suggestions and insights to develop better Rust programs. Using the results of their manual analysis, they designed two bug detectors, and provided recommendations for developing more bug detectors in the future. Our approach to finding bug patterns has similar implications to their work but, due to automation, can operate at a far larger scale.

Li et al. (2021) present MirChecker, a fully automated bug detection framework for Rust. This framework works by carrying out static analysis on Rust's Mid-level Intermediate Representation (MIR). The tool exploits the insights obtained from manually observing existing bugs (by studying reported CVEs) in Rust code bases. Using custom abstract domains that consider both numerical (e.g. integer bounds) and symbolic (e.g. modelling memory) information, the framework detects errors using constraint solving techniques. MirChecker detected 33 new bugs, including 16 memory safety faults, across 12 Rust crates. While MirChecker automatically detects bugs, the bug classes that it detects must be manually programmed into the system, and our work is complementary to MirChecker in potentially proposing novel classes of bugs that need to be fixed.

6 Conclusion

In this project, we developed Ruxanne, a tool for mining bug fix patterns in Rust. Using this tool, we mined the top 18 most starred Rust projects in

GitHub to discover common bug fix patterns within their code changes. At the heart of Ruxanne, we used a novel code embedding approach to embed the most important aspects of a code change in a fixed sized datapoint. We processed 87,726 datapoints drawn from 57,214 commits across these 18 projects. After using the DBSCAN clustering algorithm, and a subsequent manual analysis, we obtained 12 clusters of general fix patterns and 8 clusters of borrow-checker related fix patterns. We found that the most common general pattern was adding or removing fields, while the most common borrow-checker related pattern was removing a `clone()` call.

References

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2018). A general path-based representation for predicting program properties. *ACM SIGPLAN Notices*, 53(4):404–419.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019a). code2seq: Generating sequences from structured representations of code. In *Proceedings of the 2019 Conference of the Association for Computational Linguistics (ACL)*, pages 6304–6315.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. (2019b). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Arcuri, A. and Briand, L. (2011). A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1–10.

Bielik, P., Raychev, V., and Vechev, M. (2016). PHOG: Probabilistic model for code. In *International Conference on Machine Learning*, pages 2933–2942.

Campos, E. C. and Maia, M. d. A. (2019). Discovering common bug-fix patterns: A large-scale observational study. *Journal of Software: Evolution and Process*, 31(7):1–28.

Cannon, L., Elliott, R., Kirchhoff, L., Miller, J., Milner, J., Mitze, R., Schan, E., Whittington, N., Spencer, H., Keppel, D., et al. (1991). *Recommended C style and coding standards*. Pocket reference guide. Specialized Systems Consultants.

Chen, Z. and Monperrus, M. (2019). A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061*.

Collins, C. R. and Stephenson, K. (2003). A circle packing algorithm. *Computational Geometry*, 25(3):233–256.

Cotroneo, D., De Simone, L., Iannillo, A. K., Natella, R., Rosiello, S., and Bidokhti, N. (2019). Analyzing the context of bug-fixing changes in the OpenStack cloud computing platform. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 334–345. IEEE.

DeGroot, M. H. and Schervish, M. J. (2012). *Probability and statistics*. Pearson Education.

Endres, A. (1975). An analysis of errors and their causes in system programs. *IEEE Transactions on Software Engineering*, 1(1):140–149.

Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231.

Eyolfson, J. (2018). *Enforcing Abstract Immutability*. PhD thesis, University of Waterloo.

Flanagan, C. and Felleisen, M. (1998). A new way of debugging Lisp programs. In *Proceedings of Lisp Users' Group Meeting (LUGM)*.

Forrest, S., Nguyen, T., Weimer, W., and Le Goues, C. (2009). A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954.

Gopinath, R., Jensen, C., Groce, A., et al. (2015). Mutant census: An empirical examination of the competent programmer hypothesis. *Technical Report, School of EECS, Oregon State University*.

Hanam, Q., Brito, F. S. d. M., and Mesbah, A. (2016). Discovering bug patterns in JavaScript. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, pages 144–156.

Hindle, A., Barr, E. T., Gabel, M., Su, Z., and Devanbu, P. (2016). On the naturalness of software. *Communications of the ACM*, 59(5):122–131.

Hoang, T., Kang, H. J., Lo, D., and Lawall, J. (2020). CC2Vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 518–529.

Huang, W., Milanova, A., Dietl, W., and Ernst, M. D. (2012). ReIm & ReImInfer: Checking and inference of reference immutability and method purity. In *OOPSLA 2012, Object-Oriented Programming Systems, Languages, and Applications*, pages 879–896, Tucson, AZ, USA.

Islam, M. R. and Zibran, M. F. (2021). What changes in where? An empirical study of bug-fixing change patterns. *ACM SIGAPP Applied Computing Review*, 20(4):18–34.

Jeffrey, D., Feng, M., Gupta, N., and Gupta, R. (2009). Bugfix: A learning-based tool to assist developers in fixing bugs. In *2009 IEEE 17th International Conference on Program Comprehension*, pages 70–79. IEEE.

Jones, J. A. and Harrold, M. J. (2005). Empirical evaluation of the Tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282.

Klabnik, S. and Nichols, C. (2019). *The Rust programming language (Covers Rust 2018)*. No Starch Press.

Knuth, D. E. (1989). The errors of TEX. *Software—Practice and Experience*, 19(7):607–685.

Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. (2012). 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)*, pages 3–13. IEEE.

Le Goues, C., Pradel, M., and Roychoudhury, A. (2019). Automated program repair. *Communications of the ACM*, 62(12):56–65.

Li, Z., Wang, J., Sun, M., and Lui, J. C. (2021). MirChecker: Detecting bugs in rust programs via static analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2183–2196.

Lin, B., Wang, S., Wen, M., and Mao, X. (2022). Context-aware code change embedding for better patch correctness assessment. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(3):1–29.

Ling, M., Yu, Y., Wu, H., Wang, Y., Cordy, J. R., and Hassan, A. E. (2022). In Rust we trust: a transpiler from unsafe C to safer Rust. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pages 354–355.

Liu, Y., Zhang, L., and Zhang, Z. (2018). A survey of test based automatic program repair. *J. Softw.*, 13(8):437–452.

Madeiral, F., Durieux, T., Sobreira, V., and Maia, M. (2018). Towards an automated approach for bug fix pattern detection. *arXiv preprint arXiv:1807.11286*.

Martinez, M. and Monperrus, M. (2012). *Mining repair actions for guiding automated program fixing*. PhD thesis, Inria.

Martinez, M. and Monperrus, M. (2015). Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205.

Monperrus, M. (2014). "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*, pages 234–242.

Moss, S. (2021). How Dropbox pulled off its hybrid cloud transition. `https://www.datacenterdynamics.com/en/analysis/how-dropbox-pulled-off-its-hybrid-cloud-transition/`. Accessed November 21, 2022.

Naish, L., Lee, H. J., and Ramamohanarao, K. (2009). Spectral debugging with weights and incremental ranking. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 168–175. IEEE.

Nguyen, T., Weimer, W., Le Goues, C., and Forrest, S. (2009). Using execution paths to evolve software patches. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 152–153. IEEE.

Pan, K., Kim, S., and Whitehead, E. J. (2009). Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315.

Qi, Y., Mao, X., and Lei, Y. (2013). Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189. IEEE.

Qi, Y., Mao, X., Lei, Y., Dai, Z., and Wang, C. (2014). The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265.

Qin, B., Chen, Y., Yu, Z., Song, L., and Zhang, Y. (2020). Understanding memory and thread safety practices and issues in real-world Rust programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 763–779.

Raychev, V., Bielik, P., Vechev, M., and Krause, A. (2016). Learning programs from noisy data. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 761–774, New York, NY, USA. Association for Computing Machinery.

Sam, G., Cameron, N., and Potanin, A. (2017). Automated refactoring of Rust programs. In *Proceedings of the Australasian Computer Science Week Multiconference*, pages 1–9.

Spadini, D., Aniche, M., and Bacchelli, A. (2018). Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 908–911.

Tan, S. H. and Roychoudhury, A. (2015). relifix: Automated repair of software regressions. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 471–482. IEEE.

Tian, H., Tang, X., Habib, A., Wang, S., Liu, K., Xia, X., Klein, J., and Bissyandé, T. F. (2022). Is this change the answer to that problem? Correlating descriptions of bug and code changes for evaluating patch correctness. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13.

Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740.

Xie, X., Chen, T. Y., Kuo, F.-C., and Xu, B. (2013). A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):1–40.

Xu, R. and Wunsch, D. (2005). Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678.

Yang, Y., He, T., Feng, Y., Liu, S., and Xu, B. (2022). Mining Python fix patterns via analyzing fine-grained source code changes. *Empirical Software Engineering*, 27(2):1–37.

Ye, H., Gu, J., Martinez, M., Durieux, T., and Monperrus, M. (2021). Automated classification of overfitting patches with statically extracted code features. *IEEE Transactions on Software Engineering*, 48(8):2920–2938.

Zhang, Y., Chen, Y., Cheung, S.-C., Xiong, Y., and Zhang, L. (2018). An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 129–140.