Breaking Down Break-It-Fix-It: An Automated Software Repair Replication

Jason M. Pittman*†, Kira Cincotta*, and Rebecca Saul*
*Booz Allen Hamilton, McLean, VA 30332 USA
†University of Maryland Global Campus, Adelphi, USA

***Abstract*—Software quality is strongly correlated with the quantity and severity of bugs. While there are a variety of tools, techniques, and practices to aid production of robust and resilient code, low quality software is draining trillions of dollars from organizations annually. Meanwhile, debugging and fixing coding errors consumes upwards of half of developer labor. To say this situation is untenable is an understatement. Fortunately, automated software repair offers a possible solution. The literature around automated code fixing has been expanding with a variety of implementations ranging from genetic programming, code translation, and various machine learning algorithms. All report positive results, however there has not yet been a dedicated effort to measure to what extent the various implementations are generalizable. Accordingly, we sought to replicate a prominent study in the field in two parts. The first part consisted of replicating the training of the machine learning model using the source study materials. We found training to be impossible at first due to package dependencies and missing package files. However, we were able to replicate the self-repair evaluation. The results were identical to the source study. Later, using a Docker compose file obtained from the original authors, we were able to replicate BIFI model training and again match outcomes. Overall, based on the replication outcomes, we offer future leaning recommendations and ideas for future work.**

***Index Terms*—software and engineering, automated software repair, deep learning, replication.**

I. Introduction

SOFTWARE Software quality is strongly correlated with the quantity and severity of bugs. Further, software quality is commonly understood to be a measure of design practices, errors per line of code, and testing acumen Misra & Bhavsar (2003); Abuasad & Alsmadi (2012); Alves et al. (2016). As it stands, poor quality software led to losses of 2.41 trillion USD in 2022 McGuire (2022). In light of these facts, it is natural to wonder why industry simply does not produce more high quality software. In fact, there are a variety of tools, techniques, and practices to aid developers and engineers in product robust and resilient code. However, automation is limited in two critical areas: debugging and generating code repairs (i.e., patches) Tufano et al. (2019); Xia et al. (2022). Bugs and vulnerabilities alike can take days to isolate, especially in large projects Alhefdhi et al. (2020). Manual debugging and patching are notoriously tedious job tasks. Often, software developers spend upwards of 50% of their time debugging Britton et al. (2013). Moreover, such manual interventions have a nonzero probability of regressing source code or introducing new bugs and vulnerabilities.

Manuscript received May 4, 2023; Corresponding author: J. M. Pittman (email: pittman_jason@bah.com).

Automated software repair Harman (2010); Le Goues et al. (2013) is one possible solution to the burden of such manual processes. Indeed, there have been a variety of proposed software self-repair implementations such as GenProg Le Goues et al. (2013), Angelix Mechtaev et al. (2016a), and Neural Machine Translation Tufano et al. (2019) techniques. More recently, Yasunaga and Liang introduced Break-It-Fix-It (BIFI) Yasunaga & Liang (2021). This study is of particular interest for three reasons. The authors offered a detailed description of their self-repair implementation. The work also referenced a full GitHub repository containing source training data, BIFI code, and trained models. As well, Yasunaga and Liang specifically recommend future work explore to what extent BIFI generalizes to other domains. However, Xia et al. Xia et al. (2022) noted existing literature and the demonstrated self-repair paradigms therein may have limited generalizability.

One means to explore the generalizability of existing work is to reproduce the source study. Logically, a replication to verify results should precede reproduction though Plesser (2018). The process of replication involves duplicating a research study using the same techniques, materials, and experimental setup Lindsay & Ehrenberg (1993); Brooks et al. (1996); Gómez et al. (2010); Plesser (2018). The purpose is to verify the results and increase trust in their accuracy and consistency. However, replication is only feasible if the original research provides a thorough description of the materials, setup, and equipment used, enabling another researcher to carry out an identical investigation. In contrast, reproduction encompasses the act of recreating a study using various instruments, data sets, or techniques, with the objective of showing that the results are not specific to one particular implementation and can be generalized Lindsay & Ehrenberg (1993); Brooks et al. (1996); Gómez et al. (2010); Plesser (2018). Additionally, reproducing a study can also uncover any shortcomings or drawbacks in the original methodology. Accordingly, the purpose of this work is to provide a rigorous, scientific replication of BIFI Yasunaga & Liang (2021).

The rest of this work is organized into four sections. First, we offer summaries of seminal and relevant related work. Doing so establishes a conceptual framework for this study. Then, we describe our replication method and how the scientific inquiry present in this work emerged during the replication. Next, we present the qualitative and quantitative results of the replication. This is followed by our recommendations and ideas for future work.

II. Background

We need to describe three general areas of related work to properly situate this study in the literature. First, we discuss automated software repair as a field of study. In doing so, we establish a foundation for conceptualizing the dominant lines of inquiry present in the literature. Next, we expand on the specific background related to Yasunaga and Liang Yasunaga & Liang (2021). In this regard, we aim to highlight results and conclusions for comparison after the replication is completed. Last, we provide an overview of scientific replication. The overview will make clear why replication is vital to the field and how software engineering is best replicated.

A. Automated Software Repair

In order to reduce the amount of time developers spend manually identifying and patching bugs, a variety of techniques have been devised to conduct automatic software repair. Here we briefly review the three major approaches in this field: search-based repair, semantics-based repair, and learning-based repair.

1) Search-Based: Search-based approaches to automatic software repair rely on syntactic analyses of the underlying buggy program. Based on the syntax of the original code, these methods first generate large pools of candidate patches for a given bug via processes like source manipulation or modification of abstract syntax trees Ding et al. (2019). Then the search space is traversed to identify the best patch from the pool of candidates, either randomly or using heuristic or other optimization strategies, including genetic programming. A patch is presumed to be successful if it passes a suite of provided test cases. Many search-based algorithms for software repair have been proposed Le Goues et al. (2012); Liu et al. (2018); Mehne et al. (2018); Qi et al. (2014); a common challenge faced by these programs is that as the search space expands, it can become very difficult to navigate effectively Le et al. (2018).

2) Semantics-Based: In contrast to search-based approaches, semantics-based approaches begin with a semantic, rather than syntactic, analysis of the original code. These methods use information derived from evaluation against test suites and symbolic execution to establish semantic constraints, which are then used to guide the generation of possible repairs Le et al. (2018). These repairs are constructed using program synthesis techniques like template-based synthesis and component-based synthesis. While semantics-based methods offer greater precision when compared with their search-based counterparts, they are inherently limited by the power of the underlying semantic analyzers Ding et al. (2019). In addition, like search-based algorithms, semantics-based algorithms consider a patch to be correct if it passes every case in the test suite. This means that both approaches are prone to produce patches that are overfit to the test suite, which is usually incomplete. Some popular semantics-based automatic software repair programs include Angelix Mechtaev et al. (2016b) and SemFix Nguyen et al. (2013).

3) Learning-Based: In recent years, a third approach to automatic software repair has taken root, driven by the entrance of machine learning researchers to this field of study. Deep learning practices have been used both to augment and to replace components of more traditional search-based or semantics-based algorithms, and these techniques have piqued interests in both academia and industry, including inside major companies like Facebook Marginean et al. (2019) and Google Mesbah et al. (2019). Deep learning solutions are admired for their generalizability; unlike older methods, they do not require domain-specific knowledge about programming languages, error types, or common patches Namavar et al. (2021). *Break-It-Fix-It* Yasunaga & Liang (2021), the work we aim to replicate in this paper, is fundamentally a learning-based approach to the automatic program repair problem, and displays some of the flexibility just mentioned, as the original authors apply their algorithm to correct bugs in both C and Python code.

B. Break-It-Fix-It

In their paper Break-It-Fix-It: Unsupervised Learning for Program Repair Yasunaga & Liang (2021), Yasunaga and Liang attempt to address one of the most pervasive problems in the field of software repair - overfitting. Many of the currently available automated repair algorithms were trained on small datasets due to the costly nature of obtaining paired `<bad code, good code>` data, which has traditionally required manual labeling Mesbah et al. (2019). As a result, these algorithms may fail to find patterns or generalize to unseen examples Pu et al. (2016). Recognizing the limitations imposed by a scarcity of true labeled data pairs, previous authors have augmented their datasets with synthetically generated pairs, where broken code examples are produced by applying random or heuristically-guided perturbations to existing examples of good (i.e. error-free) code Gupta et al. (2017). However, the distribution of errors in synthetically-generated bad code often does not align with the distribution of errors seen in real examples of faulty code. Thus, repair algorithms trained on these synthetic datasets fail to perform well when deployed in the real world Yasunaga & Liang (2020).

Yasunaga and Liang address the shortcomings of earlier synthetic datasets in Break-It-Fix-It (BIFI) by training a *breaker* to, given good code, generate realistic examples of bad code. The synthetic dataset produced by the breaker is then used to train a software repair algorithm, or fixer. More specifically, they formulate their task and approach as follows:

Input: An unlabeled dataset of code examples D and a critic (e.g. a code analyzer or compiler) c such that for $x \in D$:

$$c(x) = \begin{cases} 1, & x \text{ has errors} \\ 0, & x \text{ has no errors.} \end{cases}$$

The critic c can be used to partition D into a set of correct code snippets and set of erroneous code snippets; that is,

$D = \{D_{\text{good}}, D_{\text{bad}}\}$. [1]

Goal: Learn a fixer f that takes a code snippet x with $c(x) = 0$ and outputs $f(x)$ such that $c(f(x)) = 1$, while minimizing the edit distance $d(x, f(x))$. [2]

Approach:

0) **[Initialize]** Start with a fixer f_0 from prior work.
1) **[Train]** Repeat the following steps for k rounds. Let i indicate the round number, starting with $i = 1$.
   a) Apply f_{i-1} to $x \in D_{\text{bad}}$. If $c(f_{i-1}(x)) = 1$, save the pair $(x, f_{i-1}(x))$. Let D_i be the dataset of (bad, corrected) code snippets produced in this step.
   b) Train a breaker b_i on D_i.
   c) Apply b_i to $x \in D_{\text{good}}$. If $c(b_i(x)) = 0$, add the pair $(b_i(x), x)$ to D_i.
   d) Train a fixer f_i on D_i
2) **[Evaluate]** Measure the accuracy of f_k on a held out set of real code snippets with errors.

The approach taken in BIFI is similar to the technique of backtranslation, in which one uses a target-to-source model to generate noisy sources, and then uses these noisy sources to train a source-to-target model Lample et al. (2017). BIFI improves on backtranslation in two primary ways: (i) BIFI uses the critic to verify that additions to D_i in steps 1a and 1c are actually$(x_{\text{wrong}}, x_{\text{correct}})$ pairs, whereas this is not guaranteed in backtranslation (ii) BIFI trains the fixer f_i on pairs of (real erroneous code, synthetic fixed good) in addition to pairs of (synthetic erroneous code, real good code), whereas backtranslation only trains the fixer on the latter set of examples.

Yasunaga and Liang assert that BIFI achieves 71.7% accuracy on DeepFix Gupta et al. (2017), a 5.6% improvement over the current state-of-the-art algorithm, as well as 90.5% accuracy on Github-Python, a dataset of three million python code snippets introduced in the original BIFI paper. Our study aims to replicate their methods and results.

C. *Scientific Replication*

Generally, in the scientific community, there is a low regard for scientific replications Lindsay & Ehrenberg (1993). As researchers often seek out new discoveries, they are inclined to wonder where the innovation lies in replicating a study exactly how it was originally conducted. However, if this were the case, if replications were truly identical to the original study, all conditions (e.g., time, testing environment) would need to be the same. Thus, all replications must involve some level of variation in the conditions. Once we accept that replication isn't merely repeating the *exact* same study, we can take advantage of the differences in the study conditions and note that despite these differences, the same results were obtained Lindsay & Ehrenberg (1993). Replication not only validates the original findings but establishes an increased range of conditions for which the findings hold, thereby extending the scope of the work Lindsay & Ehrenberg (1993); Brooks et al. (1996).

According to Gómez, Juristo, and Vegas there are five notable elements in software engineering experimentation that form the structure of an experiment and may vary in replication: Site, Experimenters, Apparatus, Operationalizations, and Population Properties. Site and Experimenters account for the experiment location and who is conducting the experiment, respectively Gómez et al. (2010). Apparatus is defined by the "experimental design, instruments, forms, materials, experimental objects and procedures used to run an experiment" Gómez et al. (2010). Operationalizations describe the independent and dependent variables that are used to measure the effects of the experimentGómez et al. (2010). Population Properties refers to the subjects and experimental objects, where subject properties are subject type and experience and experimental objects are "specifications, design documents, source codes, programs or any other artefact related to the software development" Gómez et al. (2010). We use these elements as a framework to ensure our BIFI replication research aligns with the scientific method.

III. Method

While we are interested in validating the results of the original study, the goal of our work is to establish an increased range of different conditions in which the findings of BIFI will hold. Simply put, we wish to know if it is possible to replicate the results of BIFI using the same data in a different testing environment. In seeking to replicate these results, we will also be evaluating whether the researcher's repository contains explicit enough instructions and links to source code that aid the clear replication of their study. Successful replication of BIFI will help us to determine if the results can be generalized and may create new avenues for potential work and innovation.

Literature suggests Lindsay & Ehrenberg (1993) that it is best to start with closer replications in the initial stages of replication because the more differentiated ones may not replicate successfully. Therefore, we first opted to complete a close replication in which we kept relatively all the known conditions of the study the same. However, due to issues with the dependencies we were unable to follow the documented protocol from the original experimenters Yasunaga & Liang (2021). Consequently, the replication design pivoted from one where little variance existed to one that has variance but follows the overall method of the reference experiment Gómez et al. (2010).

Overall, we sought to conduct a replication of the BIFI study to assess the generalizability of the research and the potential for further use of the self-repair function. While we came across some dependency issues at first, we were able to produce results that corresponded to the ones reported in the BIFI paper by using their trained models. However, based on the author's provided documentation, we were unable to train

[1] Note: the code snippets in D_{good} and D_{bad} have no relation to each other. We do NOT have pairs of snippets $(x_{\text{bad}}, x_{\text{good}})$ where x_{bad} is a piece of code with errors and x_{good} is a corrected, error-free version of x_{bad}.

[2] In software repair, we want the fixer f to be semantics-preserving, but since this is very difficult to check, Yasunaga and Liang use edit distance as a proxy measure, under the assumption that if the edit distance between two code snippets is small, they are semantically similar.

our own models and therefore the scope to which BIFI can be expanded may be limited. Details are provided in the next section.

IV. Results

We present the results in two sections. The first covers the replication of the BIFI Yasunaga & Liang (2021) initial fixer training. Then, in the second section, we reveal the results of evaluating the fixer with the models we trained as well as the trained models Yasunaga and Liang provided as part of the materials. Along the way, we outline critical findings and important details that contribute to answer the research question.

A. Training the BIFI fixer

We followed the steps as outlined in the BIFI GitHub repository. However, we encountered numerous errors. Some errors were correctable and we advanced to the next step. Ultimately, however, some errors were terminal. The details are as follows.

We cloned the BIFI repository Yasunaga & Liang (2021). Next, we followed the BIFI environment creation commands indicated in the repository up to `pip install -e`. Installing in editable mode produced errors because of a missing file. We corrected the error by back-tracing to the most likely version of the `fairseq` package in Facebook's GitHub and manually insert the missing files (Table I) in the `fairseq\data` directory. To back-trace, we triangulated the `fairseq` release based on the time of BIFI publication, the BIFI repository last commit timestamp, and the `fairseq` version release dates. We were able to build `fairseq` successfully after inserting these files from the Facebook repository into the local BIFI environment.

TABLE I
Fairseq file replacements and the versions

	Versions	
File	BIFI	Replaced
data_utils_fast.pyx	0.10.2	0.10.2
token_block_utils_fast..pyx	0.10.2	0.10.2
dictionary.py	0.10.2	0.10.2

Following the install of `numpy` and `editstance`, we downloaded the minimal dataset from the BIFI repository. We then created the set of *round* directories per the layout diagram Yasunaga and Liang provided. However, executing the python statements from `run-round0.sh` produced errors. In this phase of the replication, we encountered import errors for various `fairseq` subordinate packages. We traced the errors to additional missing files in the same directory as before. At this point, we opted to simply copy the Facebook `\data` subdirectory for version 0.10.2 into the BIFI local environment. While this resolved missing file issues, we then encountered a series of package import obstacles. Fixing these, while conceptually possible, would require editing BIFI source code which we elected to not do given the intended goal and methodology of this replication.

For completeness, we also tried to implement the BIFI training procedure in an updated local environment. The environment reflected a current software stack (i.e., python 3.10.6, fairseq 0.12.2, numpy 1.24.1). This was not successful due to significant differences between package versions and the BIFI source code architecture. Therefore, getting the BIFI system to a working state based on replicating training the models was not possible with the materials available in the BIFI GitHub repository.

However, one of the authors (Yasunaga) responded to an email inquiry regarding the above. The response included a Docker compose file and a `fairseq` package build from March 2020. Upon inspection, we observed six differences between the Docker compose directives, the BIFI repository instructions and contents, as well as the details in the original paper. We indexed all the components and range of versions (Table II) before attempting further replication of model training.

TABLE II
Breakdown of BIFI environmental components

	Versions	
Component	Docker	GitHub & Paper
Operating System	Ubuntu 16.04	NA[1]
CUDA	10.1	NA[1]
cuDNN	7	NA[1]
miniconda	latest	NA[1]
Python	3.7.7	3.7.7
pytorch	1.4.0	1.4.0
torchvision	0.5.0	0.5.0
tqdm	4.53.0	latest
numpy	1.20.1	1.20.1
editdistance	latest	latest
fairseq[2]	0.9.x	NA[1]

[1] NA: information is not available in the BIFI GitHub repository materials.

[2] Version information had to be inferred from file date stamps and other file object clues contrasted against package release history.

We then decided to pursue two model training paths given the updated BIFI component information. First, we used the Docker compose file to deploy a container environment as recommended by Yasunaga. Second, we took the `fairseq` package from Yasunaga and overwrote the same directory in the local GitHub repository clone. In both cases, we injected the training data as detailed by the authors. Training completed without error for both the Docker container and the local clone environments. Further, both training paths yielded models identical in file size and object metadata to the trained models provided in the BIFI repository.

B. Evaluating the BIFI fixer

We were able to run the BIFI fixer evaluation without error using all three trained models. We used the full data download from the BIFI repository. Then, we ran Yasunaga and Liang's final BIFI evaluation step against the models trained in our Docker and local repository clone environment. More specifically, we ran the `python src/c005__eval_fixer.py` routine using the `--round_name round2-BIFI-part2` option. Our intent was to replicate Yasunaga and Liang's *round-2* accuracy in Total (90.5%), for Unbalanced Parentheses (94.2%), Indentation Error (85.9%), and Invalid Syntax (93.5%). The results from our replication trials are as follows (Table III).

TABLE III
REPLICATION RESULTS

	Accuracy		
Category	Docker	Local	Yasunaga & Liang (2021)
Total	90.5%	90.5%	90.5%
Unbalanced Parentheses	94.2%	94.2%	94.2%
Indentation Error	85.9%	85.9%	85.9%
Invalid Syntax	93.5%	93.5%	93.5%

Considering the outcomes of this replication, we reached several conclusions. These are discussed in the next and final section of this work. We also detail where our assumptions and limitations differ from those offered by Yasunaga and Liang. As well, as a replication of existing work, we have unique recommendations and ideas for future research in software self-repair. In any case, we begin by offering a brief summary as a grounding mechanism for conceptualizing our results specifically and software self-repair in general.

V. CONCLUSION

Today, software developers spend upwards of 50% of their time finding and patching bugs and vulnerabilities Britton et al. (2013); Alhefdhi et al. (2020). Automated software repair Harman (2010); Le Goues et al. (2013) is intended to reduce or eliminate such a labor burden, thus freeing trapped development capacity. One current attempt at self-repair was Yasunaga and Liang's Break-It-Fix-It (BIFI) Yasunaga & Liang (2021). The authors demonstrated a self-repair function capable of repairing errors with greater than 90% accuracy. However, Yasunaga and Liang specifically recommend future work explore to what extent BIFI generalizes to other domains. Echoing such a sentiment, Xia et al. Xia et al. (2022) noted existing literature, and the demonstrated self-repair paradigms therein, may have limited generalizability.

For that reason, the purpose of this work was to conduct a rigorous, scientific replication of BIFI. We were largely successful in this effort, with one major caveat. On one hand, our replication succeeded in validating the BIFI evaluation results using just the code, documentation, and pre-trained models available in the BIFI repository published by the authors. We take such an outcome as one step completed towards assessing the generalizability of software self-repair in general and BIFI in specific. On the other hand, we were initially unable to replicate the model training protocol for BIFI due to missing files and package import errors in the public version of the code. These issues were only resolved once we contacted the BIFI authors and received a copy of the Docker compose file used to generate the environment used in their experiments. Using this compose file, we were able to construct a viable model training environment and build the models using directions in the BIFI repository [3].

[3]We did not receive permission to share the Docker compose file.

Though we eventually succeeded in training the BIFI model, and achieved the same performance the authors claim, the successful implementation of BIFI used significantly outdated versions of Python, numpy and fairseq. Furthermore, attempts to reimplement BIFI with updated versions of these packages proved fruitless. Based on this result, we surmise BIFI is not actively maintained in the software engineering sense. The last repository commit was on August 31, 2021. The technology stack (i.e., Python, fairseq, and so forth) have evolved since that commit and without the Docker compose file it is not possible to use BIFI independent of its pre-trained models.

A. Assumptions and Limitations

Of course, we have assumed the GitHub repository provided by Yasunaga and Liang Yasunaga & Liang (2021) contains the correct codebase and training data. The assumption is reasonable because the commit history is aligned with the publication date. Further, a thorough search did not uncover any additional code repositories. We also assume the errors encountered during the BIFI training procedure are unresolvable given publicly available documentation on BIFI. While it might be possible to engineer a Python environment suitable for BIFI training, the source paper lacks complete information particularly as it relates to the fairseq package, and the Docker compose file we used to resolve issues with fairseq is not published at this time. Therefore, potential resolutions for the errors are limited. On a related point, we recognize the limitation of our results in establishing generalizability for BIFI. Achieving identical results, while positive, do not fully establish external, general application of the automatic software self-repair tool.

B. Recommendations

Given the challenges that emerged during the replication effort, it is important that researchers consider the pace at which technology evolves. Research that provides clear, executable documentation and timeless code will help to ensure that future technology can build on existing work. Therefore, our BIFI replication work should be extended to explore the technology debt (tech debt) that exists in current research. Briefly, tech debt speaks to the measurable value associated with short term technology design or implementation decisions in exchange for increased maintenance costs long term. At an extreme, tech debt is highest when no maintenance takes place or technology is developed and abandoned. Thus, if the aim of self-repair functions is to alleviate the time spent by developers on identifying and patching bugs, we must first look at the sustainability of the research in this area.

A first step to doing this would be to find the percentage of relevant papers (with code) that are in a tech debt state. From here, future work should explore which of this work has been updated and if the patches stay true to the original functionality of the code. Similar to the Docker file shared by Yansunaga, practitioners and researchers alike should consider using a container environment to better preserve their work and increase the likelihood of successful replication. Utilizing a container environment enables developers to share code and its dependencies with others, thereby reducing the potential for errors and hopefully slowing down the pace at which the research enters a state of technical debt. On this note, it may be worthwhile to examine how many self-repair function papers utilize micro-server architectures. Once we have an understanding for the ubiquity of tech debt in recent self-repair work, it can be extended into future research such as self-repair functions that automatically review code and are built into the memory environment of the workspace. Such innovation will only evolve if we write code with tomorrow in mind so that our systems of today can be used in the future. Finally, it may be of value to the field if future work is able to reproduce the BIFI accuracy measures using an updated technology stack. Doing so will invariably require changes to the BIFI source code but will add to the body of evidence for generalizability of the self-repair function.

References

Abuasad, A., & Alsmadi, I. M. (2012). Evaluating the correlation between software defect and design coupling metrics. In *2012 international conference on computer, information and telecommunication systems (cits)* (pp. 1–5).

Alhefdhi, A., Dam, H. K., Le, X.-B. D., & Ghose, A. (2020). Adversarial patch generation for automatic program repair. *arXiv preprint arXiv:2012.11060*.

Alves, H., Fonseca, B., & Antunes, N. (2016). Software metrics and security vulnerabilities: dataset and exploratory study. In *2016 12th european dependable computing conference (edcc)* (pp. 37–44).

Britton, T., Jeng, L., Carver, G., & Cheak, P. (2013). Reversible debugging software "quantify the time and cost saved using reversible debuggers". *University Cambridge: Cambridge, UK*.

Brooks, A., Daly, J., Miller, J., Roper, M., & Wood, M. (1996). Replication of experimental results in software engineering. *International Software Engineering Research Network (ISERN) Technical Report ISERN-96-10, University of Strathclyde*, *2*.

Ding, Z. Y., Lyu, Y., Timperley, C., & Le Goues, C. (2019). Leveraging program invariants to promote population diversity in search-based automatic program repair. In *2019 ieee/acm international workshop on genetic improvement (gi)* (p. 2-9). doi: 10.1109/GI.2019.00011

Gómez, O. S., Juristo, N., & Vegas, S. (2010). Replications types in experimental disciplines. In *Proceedings of the 2010 acm-ieee international symposium on empirical software engineering and measurement* (pp. 1–10).

Gupta, R., Pal, S., Kanade, A., & Shevade, S. (2017, Feb.). Deepfix: Fixing common c language errors by deep learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, *31*(1). Retrieved from `https://ojs.aaai.org/index.php/AAAI/article/view/10742` doi: 10.1609/aaai.v31i1.10742

Harman, M. (2010). Automated patching techniques: the fix is in: technical perspective. *Communications of the ACM*, *53*(5), 108–108.

Lample, G., Conneau, A., Denoyer, L., & Ranzato, M. (2017). *Unsupervised machine translation using monolingual corpora only.* arXiv. Retrieved from `https://arxiv.org/abs/1711.00043` doi: 10.48550/ARXIV.1711.00043

Le, X.-B. D., Thung, F., Lo, D., & Goues, C. L. (2018). Overfitting in semantics-based automated program repair. In *Proceedings of the 40th international conference on software engineering* (p. 163). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3180155.3182536` doi: 10.1145/3180155.3182536

Le Goues, C., Forrest, S., & Weimer, W. (2013). Current challenges in automatic software repair. *Software quality journal*, *21*(3), 421–443.

Le Goues, C., Nguyen, T., Forrest, S., & Weimer, W. (2012). Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, *38*(1), 54-72. doi: 10.1109/TSE.2011.104

Lindsay, R. M., & Ehrenberg, A. S. (1993). The design of replicated studies. *The American Statistician*, *47*(3), 217–228.

Liu, K., Koyuncu, A., Kim, K., Kim, D., & F. Bissyandé, T. (2018). Lsrepair: Live search of fix ingredients for automated program repair. In *2018 25th asia-pacific software engineering conference (apsec)* (p. 658-662). doi: 10.1109/APSEC.2018.00085

Marginean, A., Bader, J., Chandra, S., Harman, M., Jia, Y., Mao, K., ... Scott, A. (2019). Sapfix: Automated end-to-end repair at scale. In *2019 ieee/acm 41st international conference on software engineering: Software engineering in practice (icse-seip)* (p. 269-278). doi: 10.1109/ICSE-SEIP.2019.00039

McGuire, M. (2022, Dec). *What is the cost of poor software quality in the u.s.?* `https://www.synopsys.com/blogs/software-security/poor-software-quality-costs-us/`. (Accessed: 11-01-2023)

Mechtaev, S., Yi, J., & Roychoudhury, A. (2016a). Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering* (pp. 691–701).

Mechtaev, S., Yi, J., & Roychoudhury, A. (2016b). Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 ieee/acm 38th international conference on software engineering (icse)* (p. 691-701). doi: 10.1145/2884781.2884807

Mehne, B., Yoshida, H., Prasad, M. R., Sen, K., Gopinath, D., & Khurshid, S. (2018). Accelerating search-based program repair. *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 227-238.

Mesbah, A., Rice, A., Johnston, E., Glorioso, N., & Aftandilian, E. (2019). Deepdelta: Learning to repair compilation errors. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (p. 925–936). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/3338906.3340455` doi: 10.1145/3338906.3340455

Misra, S. C., & Bhavsar, V. C. (2003). Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *International conference on computational science and its applications* (pp. 724–732).

Namavar, M., Nashid, N., & Mesbah, A. (2021). *A controlled experiment of different code representations for learning-based bug repair.* arXiv. Retrieved from `https://arxiv.org/abs/2110.14081` doi: 10.48550/ARXIV.2110.14081

Nguyen, H. D. T., Qi, D., Roychoudhury, A., & Chandra, S. (2013). Semfix: Program repair via semantic analysis. In *2013 35th international conference on software engineering (icse)* (p. 772-781). doi: 10.1109/ICSE.2013.6606623

Plesser, H. E. (2018). Reproducibility vs. replicability: a brief history of a confused terminology. *Frontiers in neuroinformatics*, *11*, 76.

Pu, Y., Narasimhan, K., Solar-Lezama, A., & Barzilay, R. (2016). *sk_p: a neural program corrector for moocs.* arXiv. Retrieved from `https://arxiv.org/abs/1607.02902` doi: 10.48550/ARXIV.1607.02902

Qi, Y., Mao, X., Lei, Y., Dai, Z., & Wang, C. (2014). The strength of random search on automated program repair. In *Proceedings of the 36th international conference on software engineering* (p. 254–265). New York, NY, USA: Association for Computing Machinery. Retrieved from `https://doi.org/10.1145/2568225.2568254` doi: 10.1145/2568225.2568254

Tufano, M., Pantiuchina, J., Watson, C., Bavota, G., & Poshyvanyk, D. (2019). On learning meaningful code changes via neural machine translation. In *2019 ieee/acm 41st international conference on software engineering (icse)* (pp. 25–36).

Xia, C. S., Wei, Y., & Zhang, L. (2022). Practical program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179*.

Yasunaga, M., & Liang, P. (2020). *Graph-based, self-supervised program repair from diagnostic feedback.* arXiv. Retrieved from `https://arxiv.org/abs/2005.10636` doi: 10.48550/ARXIV.2005.10636

Yasunaga, M., & Liang, P. (2021). Break-it-fix-it: Unsupervised learning for program repair. In *International conference on machine learning* (pp. 11941–11952).