# How Effective Are Neural Networks for Fixing Security Vulnerabilities

Yi Wu
Purdue University
West Lafayette, USA
wu1827@purdue.edu

Nan Jiang
Purdue University
West Lafayette, USA
jiang719@purdue.edu

Hung Viet Pham*
York University
Toronto, Canada
hvpham@yorku.ca

Thibaud Lutellier*
University of Alberta
Camrose, Canada
lutellie@ualberta.ca

Jordan Davis
Purdue University
West Lafayette, USA
davi1304@purdue.edu

Lin Tan
Purdue University
West Lafayette, USA
lintan@purdue.edu

Petr Babkin
J.P. Morgan AI Research
Palo Alto, USA
petr.babkin@jpmorgan.com

Sameena Shah
J.P. Morgan AI Research
New York, USA
sameena.shah@jpmchase.com

## ABSTRACT

Security vulnerability repair is a difficult task that is in dire need of automation. Two groups of techniques have shown promise: (1) large code language models (LLMs) that have been pre-trained on source code for tasks such as code completion, and (2) automated program repair (APR) techniques that use deep learning (DL) models to automatically fix software bugs.

This paper is the first to study and compare Java vulnerability repair capabilities of LLMs and DL-based APR models. The contributions include that we (1) apply and evaluate five LLMs (Codex, CodeGen, CodeT5, PLBART and InCoder), four fine-tuned LLMs, and four DL-based APR techniques on two real-world Java vulnerability benchmarks (Vul4J and VJBench), (2) design code transformations to address the training and test data overlapping threat to Codex, (3) create a new Java vulnerability repair benchmark VJBench, and its transformed version VJBench-trans, to better evaluate LLMs and APR techniques, and (4) evaluate LLMs and APR techniques on the transformed vulnerabilities in VJBench-trans.

Our findings include that (1) existing LLMs and APR models fix very few Java vulnerabilities. Codex fixes 10.2 (20.4%), the most number of vulnerabilities. Many of the generated patches are un-compilable patches. (2) Fine-tuning with general APR data improves LLMs' vulnerability-fixing capabilities. (3) Our new VJBench reveals that LLMs and APR models fail to fix many Common Weakness Enumeration (CWE) types, such as CWE-325 Missing cryptographic step and CWE-444 HTTP request smuggling. (4) Codex still fixes 8.7 transformed vulnerabilities, outperforming all the other LLMs

and APR models on transformed vulnerabilities. The results call for innovations to enhance automated Java vulnerability repair such as creating larger vulnerability repair training data, tuning LLMs with such data, and applying code simplification transformation to facilitate vulnerability repair.

## CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Automatic programming**; • **Computing methodologies** → *Neural networks*; • **Security and privacy** → *Software security engineering*.

## KEYWORDS

Automated Program Repair, Large Language Model, Vulnerability, AI and Software Engineering

## 1 INTRODUCTION

Software vulnerabilities, such as buffer overflows and SQL injections, have a critical impact on global economies and can harm millions of users. Once a vulnerability is discovered, it is often crucial to fix it promptly to minimize the potential for exploitation. Yet, recent studies [43, 52] find that the average time to fix a vulnerability (time between the discovery and the fix) varies between 60 to 79 days, which is still too long and provides ample opportunities for attackers to exploit these vulnerabilities. For example, for the severe Apache Log4Shell vulnerability reported on November 24, 2021, the first fix was deployed by Apache 12 days after the report. During these 12 days, both Cloudflare and Cisco reported several attacks exploiting the vulnerability [34]. Moreover, the initial fix proved insufficient, leaving Log4Shell vulnerable until a complete

Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah

fix was released more than one month later. As a result, there is a need for faster vulnerability-fixing solutions.

Most vulnerability benchmarks and vulnerability repair solutions focus either on C/C++ [19, 29–31, 36, 42, 46, 53, 67] or binaries [10, 48, 54, 60, 72]. There is a lack of solutions and benchmarks for Java, despite it being a widely-used programming language (the third most popular language in the open-source community [32]) with *many severe vulnerabilities*.

Java has been used to implement important servers, including web servers and services (e.g., Tomcat, Spring, CFX, Log4J), which are especially vulnerable to attackers. Consequently, many of the most critical vulnerabilities are in Java software. For example, Google assessed that the Log4Shell vulnerability in the Log4J package affected 17,000 Maven projects [7], and Microsoft even reported that nation-state attackers exploited the vulnerability [2].

Benchmarks and solutions for other programming languages often do not work or work poorly for fixing Java vulnerabilities. For example, the most common vulnerabilities in C/C++ are buffer overflows [24, 59]. Java, as a type-safe language, is designed to avoid buffer overflows. Thus, most C/C++ techniques focusing on buffer overflow vulnerabilities are irrelevant to Java. We need new benchmarks and techniques for fixing Java security vulnerabilities.

Instead of building a technique to fix Java vulnerabilities automatically, we study and compare the space and feasibility of applying two types of techniques—learning-based automated program repair and LLMs—to fix Java security vulnerabilities automatically. First, learning-based program repair has gained popularity [18, 21, 22, 40, 47, 75, 75, 76, 76]. These encoder-decoder approaches learn from a large number of pairs of bugs and their fixes (in open-source projects) to fix unseen Java software bugs automatically. *It would be interesting to study how effective such learning-based program repair models are in fixing a subset of software bugs, i.e., software vulnerabilities.*

Secondly, LLMs have recently been applied to source code [17, 25, 37, 40, 50, 63, 73] and are pre-trained models that have been trained on a tremendous amount of source code (e.g., the entirety of GitHub). Different from APR models, pre-trained LLMs learn from large corpus of source code (instead of pairs of bugs and their fixes) for various tasks such as identifier tagging and code completion. Despite learning to perform tasks different from repairing, recent study [38, 74] shows that pre-trained LLMs have competitive capabilities of fixing general Java bugs [41, 44]. *It would be interesting to study how effective such LLMs are for a different task, i.e., fixing software vulnerabilities, when they do not see how bugs are fixed.*

Thirdly, it would be interesting to compare deep learning (DL)-based APR techniques' and LLMs' capabilities of fixing Java vulnerabilities. DL-based APR techniques and LLMs represent two angles of applying models for a different task. Applying DL-based APR techniques to fix vulnerabilities is using models learned from a general dataset for a specific subset of the dataset (software vulnerability is a type of software bug). Applying LLMs to fix vulnerabilities is using models learned from a different format of dataset (sequences of code) for another format (pairs of buggy and fixed code). Since LLMs do not require pairs of bugs and their fixes, LLMs are typically built from data that is orders of magnitude larger than the training data used to train APR models. *Would more data win or data-format matching win?*

Lastly, pre-trained LLMs are often fine-tuned to adapt to different downstream tasks [8, 26, 33, 65, 73]. A recent study [38] shows that fine-tuning improves LLMs' fixing capabilities by at least 31%. However, given the lack of Java vulnerability data, it is unrealistic to fine-tune LLMs for fixing Java vulnerabilities. Thus, *it would be interesting to study how effective LLMs fine-tuned with general APR data are in fixing software vulnerabilities.* And when compared with DL-based APR techniques, *would more data plus fine-tuning win or data-format matching win?*

## 1.1 Our Approach

We conduct the first study to evaluate and compare APR techniques' and LLMs' abilities of fixing Java vulnerabilities. We evaluate five LLMs (Codex [1], CodeT5 [73], CodeGen [55], PLBART [8] and InCoder [28]), four LLMs that are fined-tuned with general APR data, and four APR techniques (CURE [40], Recoder [76], RewardRepair [75], and KNOD [39]) on two Java vulnerability benchmarks (Vul4J and a new VJBench that we create). There are two main challenges.

First, there are few benchmarks available for evaluating Java vulnerability repair tools. While Vul4J [16] contains 79 reproducible Java vulnerabilities, they belong to only 25 CWEs, i.e., types of vulnerabilities. In addition, 60% of the CWEs in the dataset (15 types of vulnerabilities) are covered by only a single reproducible vulnerability.

To address this challenge, we develop new benchmarks. We analyze the entire National Vulnerability Database (NVD) [4] to identify reproducible real-world Java vulnerabilities that are suitable for vulnerability repair evaluation, and use these to create our VJBench benchmark. These vulnerabilities cover an additional twelve CWE types not included by the Vul4J dataset and add more vulnerabilities to four CWE types with which Vul4J has only one vulnerability associated. The new benchmark can facilitate the evaluation of future Java vulnerability repair techniques.

The second challenge arises from the fact that Codex was trained on a substantial code corpus collected from GitHub [17] and the training dataset is unreleased. Since the projects in Vul4J and VJBench are public repositories on GitHub, one cannot be certain that the vulnerabilities in Vul4J and VJBench are not in Codex's training data. This is a major known threat to the validity of evaluation [11, 69]. While dataset HumanEval [17] is not in Codex's training data, it is for Python code completion and does not contain Java vulnerabilities. Creating new real-world benchmarks is not only expensive [16, 41], but might also be impracticable if LLMs have been trained on all public datasets.

Our best-effort solution to mitigate this challenge is to transform the vulnerability code in existing benchmarks. We use two types of code transformation: identifier renaming and code structure change. These transformations generate new equivalent programs that still retain the vulnerabilities but are not included in any open-source dataset that Codex and other LLMs may have seen. As a result, we create VJBench-trans, a benchmark of transformed vulnerabilities, by applying two transformation strategies on vulnerabilities from Vul4J and VJBench.

## 1.2 Contributions

Our paper makes the following contributions:

- We conduct the first study that evaluates the fixing capabilities of five LLMs, four fine-tuned LLMs, and four APR techniques on real-world Java vulnerabilities from two benchmarks Vul4J and our new VJBench. Our findings include:
  - Existing LLMs and APR techniques fix very few Java vulnerabilities. Codex fixes 10.2 (20.4%) vulnerabilities on average, exhibiting the best fixing capability. (Section 6.1)
  - Fine-tuning with general APR data improves LLMs' vulnerability-fixing capabilities. Fine-tuned InCoder fixes 9 vulnerabilities, exhibiting competitive fixing capability to Codex's. (Section 6.1)
  - Codex has the highest compilation rate of 79.7%. Other LLMs (fine-tuned or not) and APR techniques have low compilation rates (the lowest being 6.4% with CodeT5 and the rest between 24.5% to 65.2%), showing a lack of syntax domain knowledge. (Section 6.1)
  - LLMs and APR models, except Codex, only fix vulnerabilities that require simple changes, such as a single deletion or variable/method replacement. (Section 6.2)
  - Our new VJBench reveals that LLMs and APR models fail to fix many CWE types including CWE-172 Encoding error, CWE-325 Missing cryptographic step, CWE-444 HTTP request smuggling, CWE-668 Exposure of resource to wrong sphere, and CWE-1295 Debug messages revealing unnecessary information. (Section 6.2)
- We create two Java vulnerability benchmarks for automated program repair: (1) *VJBench*, which contains 42 reproducible real-world Java vulnerabilities that cover twelve new CWE types, and (2) *VJBench-trans*, which contains 150 transformed Java vulnerabilities.
- We use code transformations to mitigate the threat that LLMs and black-box Codex may have seen the evaluated benchmarks.
- We evaluate LLMs and APR techniques' fixing capabilities on transformed vulnerabilities (VJBench-trans).
  - Code transformations make LLMs and APR techniques fix fewer number of vulnerabilities. Some models such as Codex and fine-tuned CodeT5 are more robust to code transformations. On the other hand, some transformations make the vulnerabilities easier to fix. (Section 6.3)
- We provide implications and suggestions for future directions (Section 6).

## 2 NEW BENCHMARK OF JAVA VULNERABILITIES

A Java APR benchmark must contain reproducible Java vulnerabilities with test cases exposing the vulnerabilities. While there is an abundance of such benchmarks for Java bugs, including Defects4J [41], QuixBugs [44], Bugs.jar [66], and Bears [49], the only Java vulnerability benchmark for APR is Vul4J [16]. Vul4J contains 79 vulnerabilities from 51 projects covering 25 CWE types. Despite a valuable first step, Vul4J offers limited coverage of CWE categories as explained in Introduction. In addition, only 35 of these vulnerabilities are applicable for evaluating state-of-the-art learning-based
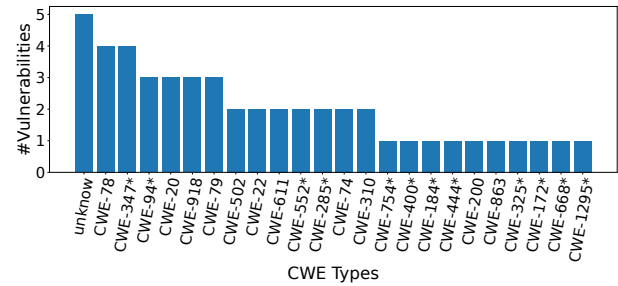


**Figure 1: CWE Type Distribution of VJBench (\* denotes the new CWE types not included in Vul4J).**

APR systems [40, 75, 76] since these APR models only fix single-hunk bugs. Specifically, 39 of the 79 vulnerabilities are single-hunk. We can only reproduce 35 of the 39 vulnerabilities, as two bugs fail to compile, and two bugs are not reproducible with the Docker container provided by the Vul4J authors.

To extend this benchmark, we collect Java vulnerabilities following prior work [41]: i) The vulnerability should only be related to Java source code, ii) The fixing commit should contain at least one test case that passes on $V_{fix}$ but fails on $V_{bug}$, iii) The fixing patch should only include changes that fix the vulnerability and should not introduce unrelated changes such as features or refactoring, and iv) the vulnerability is not already in Vul4J.

We download all available vulnerability data in JSON format on May 13, 2022 from NVD. We parse this data and obtain a list of 7,116 GitHub projects by collecting the reference URLs of these vulnerabilities. We exclude projects which have less than 50% of their code in Java, resulting in 400 Java projects containing 933 unique vulnerabilities. We then try to identify the fixing commits for each of the 933 vulnerabilities by manually checking the reference links provided in the vulnerability report or by searching the vulnerability ID in the GitHub repository if no link is provided. We find vulnerability-fixing commits for 698 vulnerabilities. Then we manually filter out 185 vulnerabilities whose fixing commits contain non-Java changes and 314 vulnerabilities that do not have test cases in their fixing commits. We now have 199 vulnerabilities, each with test cases and a corresponding Java-only fixing commit. We then successfully reproduce 42 Java vulnerabilities that are not included in Vul4J, using building tools such as Maven or Gradle.

We end up with a dataset of **42 new reproducible real-world Java vulnerabilities** from thirty open-source projects. In detail, our dataset consists of *27 multi-hunk vulnerabilities* from twenty-two projects and *15 single-hunk vulnerabilities* from eleven projects. As Figure 1 shows, these 42 vulnerabilities covers a total of 23 CWE types. Furthermore, our dataset introduces **12 new CWE types** (denoted by \* in Figure 1) not included in Vul4J and supplements four CWE types (CWE-78, CWE-200, CWE-310, CWE-863) for which Vul4J only has one example.

Table 1 describes the 15 new single-hunk vulnerabilities of twelve CWE types in our *VJBench* benchmark. There are six new unique CWE types of vulnerabilities not present in Vul4J. As a result, there are 15 vulnerabilities from VJBench and 35 vulnerabilities from Vul4J, a total of **50 vulnerabilities** that we use in our study.

**Table 1: List of the 15 new single-hunk vulnerabilities categorized by their corresponding CWE. The vulnerability IDs compose of the project name and the bug index. * denotes the six new CWE types that our benchmark adds compared to Vul4J. Jenkins-1 and Flow-2 both belong to two CWE categories.**

| CWE | Description | Vulnerability IDs |
|---|---|---|
| 20 | Improper Input Validation | Pulsar-1 |
| 22 | Improper limitation of path name to a restricted directory | Halo-1 |
| 74 | Improper Neutralization of Elements in Output ('Injection') | Ratpack-1 |
| 79 | Cross-site Scripting | Json-sanitizer-1 |
| 172* | Encoding error | Flow-1 |
| 200 | Exposure of sensitive information | Jenkins-1, Jenkins-2, Jenkins-3 |
| 325* | Missing cryptographic step | Jenkins-1 |
| 347* | Improper Verification of Cryptographic Signature | BC-Java-1 |
| 444* | HTTP request smuggling | Netty-1, Netty-2 |
| 611 | Improper restriction of XML external entity reference | Quartz-1, Retrofit-1 |
| 668* | Exposure of resource to wrong sphere | Flow-2 |
| 1295* | Debug messages revealing unnecessary information | Flow-2 |
| *unk* | no specific CWE category | Jinjava-1 |

**Table 2: Input Formats of Large Language Models**

| Model | Input Format |
|---|---|
| Codex | Comment buggy lines (BL) with hint "BUG:" and "FIXED:" Prefix prompt: Beginning of the buggy function to BL comment Suffix prompt: Line after BL comment to end of the buggy function |
| CodeT5 | Mask buggy lines with <extra_id_0> and input the buggy function |
| CodeGen | Input beginning of the buggy method to line before buggy lines |
| PLBART | Mask buggy lines with <mask> and input the buggy function |
| InCoder | Mask buggy lines with <mask> and input the buggy function |
| Tuned LLMs | Comment buggy lines and input the buggy function |

## 3 LARGE LANGUAGE MODELS AND APR TECHNIQUES

### 3.1 Large Language Models

We select five LLMs, i.e., Codex, PLBART, CodeT5, CodeGen and InCoder, because they are (1) state-of-the-art, (2) capable of performing code generation tasks without any modifications to the models or additional components (e.g., CodeBERT [26] GraphCode-BERT [33] are excluded), and (3) trained with enough source code so that they can understand code to some extent (e.g., we exclude T5 [65], GPT-2 [64], GPT-Neo [13] and GPT-J [71], whose training data is over 90% text). In this work, we study the LLMs in two settings: as is and fine-tuned with general APR data.

*3.1.1 Large Language Models As Is.* In this section, we introduce the details of the studied LLMs and how to use them for fixing vulnerabilities. Table 3 provides the model sizes and their training data information.

**Codex [17]:** Codex is a GPT-3-based [15, 17] language model with 12B parameters trained on both natural language and source code. We use the davinci-002 model (as of July 2022), which is supposed to be the most accurate Codex model [1]. We focus on Codex's

```
Prefix:
private int extend(int v, final int t){ ...
    /* BUG:
     * while (v < vt) {
     * FIXED:
     */
Suffix:
    vt = (-1 << t) + 1; ... }
Expected Output:
    if (v < vt) {
```

**Figure 2: An example input to Codex and its expected output**

insertion mode as it provided the best results in our preliminary study among the three main modes: completion, insertion, and edit.

**CodeT5 [73]:** CodeT5 is an encoder-decoder transformer model [70] pre-trained with an identifier-aware denoising objective and with bimodal dual generation tasks. It is trained on a corpus of 5.2 million code functions and 8.3 million natural language sentences from open-source repositories in six programming languages including Java. In this work, we use the largest CodeT5 model released, which has 770M parameters.

**CodeGen [55]:** CodeGen models are a series of autoregressive decoder-only transformers trained for conversational program synthesis. Their training data consists of 354.7B natural language tokens from THEPILE dataset and 150.8B programming language tokens extracted from a subset of the Google BigQuery database. In this work, we apply the CodeGen model which contains 6B parameters (the larger model with 16B parameters is not used due to the limitation of our machine).

**PLBART [8]:** PLBART uses an encoder-decoder transformer architecture with an additional normalization layer on the encoder and decoder. It's pre-trained on functions extracted from Java and Python GitHub repositories via denoising autoencoding. Two PLBART models of different sizes are available, and we use the larger model containing 400M parameters.

**InCoder [28]:** InCoder models follow XGLM [45]'s decoder-only architecture and are pre-trained on the masked span prediction task. Its pre-training data comes from open-sourced projects on GitHub and GitLab, and StackOverflow posts. There are two InCoder models of different sizes released, and we use the larger one which contains 6B parameters.

**Input Formats:** Table 2 illustrates the input format we used for each model. For Codex, we adopt an input format similar to the one used in prior work [58]. The prompt includes the commented buggy code with hint words "BUG:" and "FIXED:" to signify the location of the bug and to guide Codex towards generating a fixed version of the code. If the number of input tokens exceeds the maximum number for a model, we truncate the code and input the code around the buggy lines. Since it is unclear how the commented buggy line prompts will affect the models' fixing capabilities, we experiment with the input with and without commented buggy lines for each model. Figure 2 shows an example of the input and expected output of Codex with buggy lines commented by /* BUG .. FIXED */.

*3.1.2 Fine-Tuned Large Language Models.* We also study the fixing capabilities of fine-tuned LLMs, since fine-tuning is a common technique to adapt a pre-trained LLM to a specific downstream task, such as code summarization or code translation [26, 28, 65, 73].

**Table 3: Model size (number of parameters) and training data size of the five LLMs we apply and report in this work**

|  |  | Codex | CodeT5 | CodeGen | PLBART | InCoder |
|---|---|---|---|---|---|---|
| #Parameters |  | 12B | 770M | 6B | 400M | 6B |
| Training Data | NL | 45.0TB | - | 1.1TB | 79.0GB | 57.0GB |
| Raw Size | PL | 159.0GB | - | 436.3GB | 576.0GB | 159.0GB |
| Training Data | NL | 499.0B | - | 354.7B | 6.7B | - |
| #Tokens | PL | 100.0B | - | 150.8B | 64.4B | - |
| Training Data | NL | - | 5.2M | - | 47.0M | - |
| #Instances | PL | - | 8.3M | - | 680.0M | - |

However, due to the lack of vulnerabilities as fine-tuning data, we use the LLMs fine-tuned with general APR data, shared by existing work [38]. Prior work [38] fine-tuned LLMs with a training dataset containing 143,666 instances collected from open-source GitHub Java projects [76]. Each data instance is a pair of buggy code and fixed code. In detail, [38] used the Adam optimizer with a learning rate of $1e^{-5}$, set batch size to one and fine-tuned for one epoch. The fine-tuned LLMs are supposed to be adjusted to vulnerability fixing task to some extent due to the similarity between vulnerability fixing and general bug fixing. We perform a search and confirm that none of the vulnerabilities we study in this work is present in the APR training data used to fine-tune the LLMs.

We cannot fine-tune Codex, since it does not offer any fine-tuning API and there is also no fine-tuned Codex available. The last row of Table 2 describes the input format for using fine-tuned LLMs, where the buggy lines are given as commented lines, and the entire function is input into the fine-tuned LLMs to generate the patched lines [38].

## 3.2 APR Techniques

We select four state-of-the-art learning-based APR techniques trained for Java bugs. These APR techniques need to be open-sourced so that we can run them on our new vulnerability benchmarks.

**CURE [40]** applies a small language model (pre-trained with 4.04M code instances) to the CoCoNuT's [47] encoder-decoder architecture to learn code syntax and propose a new code-aware strategy to remove invalid identifiers and increase the compilation rate during inference. CURE is trained with 2.72M APR instances.

**Recoder [76]** uses an tree-based deep learning network that is trained on 82.87K APR training instances. It focuses on generating edits to modify buggy ASTs to form the patched ASTs.

**RewardRepair [75]** includes compilation in the calculation of the model's loss function to increase the number of compilable (and correct) patches. This is different from CURE as the loss function increases the number of compilable patches during training. Overall, RewardRepair is trained with 3.51M APR training instances.

**KNOD [39]** proposes a novel three-stage tree decoder to generate the patched ASTs, and also uses domain-knowledge distillation to modify the loss function to let the models learn code syntax and semantics. KNOD is trained with 576K APR training instances, and is the state-of-the-art DL-based APR techniques.

```java
public static void checkDirectoryTraversal(...) {
    ...
    if (pathToCheck.startsWith(parentPath.normalize())) { ... }
    throw new ForbiddenException(...); ... }
```

**(a) Before identifier renaming**

```java
public static void examineUnauthorizedPathAccess(...) {
    ...
    if (examinePath.startsWith(basePath.normalize())) { ... }
    throw new ProhibitedException(...); ... }
```

**(b) After identifier renaming**

**Figure 3: Identifier renaming for Halo-1. Functions "startsWith" and "normalize" remain intact as they are Java library functions.**

```java
if (!(value.getClass().equals(String.class)) || ...)
```

**(a) Before function chaining**

```java
Class value_class = value.getClass()
if (!(value_class.equals(String.class)) || ...)
```

**(b) After function chaining**

**Figure 4: Function chaining for VUL4J-30**

```java
if (pathToCheck.startsWith(parentPath.normalize())) {...}
```

**(a) Before function-argument passing**

```java
Path normalizedParentPath = parentPath.normalize();
if (pathToCheck.startsWith(normalizedParentPath)) {...}
```

**(b) After function-argument passing**

**Figure 5: Function-argument passing for Halo-1.**

## 4 CODE TRANSFORMATION

To address the challenge of training-testing data overlap, we need to create vulnerabilities and their fixes that have not been seen by existing LLMs or APR techniques. We generate unseen vulnerabilities by transforming existing vulnerabilities to their semantically equivalent forms. None of the APR models and LLMs, including Codex, have seen these transformed buggy code and the corresponding fixes in their training set. We apply two categories of transformations to Vul4J and VJBench, which are described below:

**(1) Identifier Renaming:** To prevent LLMs and APR models from simply memorizing the exact correct patches associated with identifier names, we rename identifiers in the buggy code and the corresponding fixed code. All variables, functions, and classes defined in the project are renamed using synonyms for the original identifier names according to Java specifications. We use synonyms to keep the word meaning of the original identifiers. We do not rename identifiers from external libraries or default Java class libraries, since one often cannot modify external libraries. Figure 3 shows an example of identifier renaming for Halo-1.

We first use the tool src2abs [6] to extract all variable, function, and class names in the buggy function, and filter out those identifiers from Java or third-party libraries. We tokenize each identifier based on camel case or snake case conventions, then use NLTK WordNet [3] to generate synonyms for each word. After that, we reassemble these synonyms to form a complete identifier. We manually review and adjust the synonyms to ensure they fit the

code context. Since some APR techniques need to extract identifiers from the whole project, we rename the identifiers used in the buggy function across the entire project.

**(2) Code Structure Change:** We define six transformation rules to change code structures.

- **If-condition flipping:** negates an if-condition and swaps the code blocks in the `if` and `else` branches.

- **Loop transformation:** converts a `for` loop to a `while` loop and vice versa.

- **Conditional-statement transformation:** turns a ternary expression (`var = cond ? exprTrue: exprFalse;`) into an `if-else` statement (`if (cond) {var = exprTrue;} else {var = exprFalse;}`), and transform a switch statement into multiple if and elseif statements, and vice versa.

- **Function chaining:** merges multiple function invocations into one call chain, or conversely splits a function call chain into separate function invocations. Figure 4 shows an example where `value.getClass().equals(...);` is split into `Class value_class = value.getClass();` and `value_class.equals(...);`.

- **Function-argument passing:** If a locally defined variable or object is only used as a function argument, we replace the function argument with its definition statement, or we extract the function call that is passed as a function argument into a separate variable/object definition. Figure 5 shows an example where the argument `parentPath.normalize()` is extracted and declared as a local object `normalizedParentPath`.

- **Code-order change:** alters the order of statements if changing the order does not affect the execution results. For example, `funcA(); int n =0;` can be transformed into `int n = 0; funcA();` as invoking `funcA()` and declaring `int n` do not affect each other.

For code structure change, we manually transform the buggy function. For each buggy function, we apply all applicable transformations at once. We further confirm the equivalence of the transformed bug by reproducing them using the same test set and applying semantically equivalent patches to pass the tests.

**A new benchmark (VJBench-trans):** In summary, to create bugs and patches that LLMs have not seen in their training set, we apply three sets of transformations (identifier renaming only, code structure change only, and both at the same time) to VJBench and Vul4J, and create *VJBench-trans* that contains 3×50 = 150 transformed Java vulnerabilities. We search in GitHub and Google the transformed code, and find no public code that is the same as the transformed buggy function.

**Recover patches for evaluation:** The transformed code is still realistic and human-readable. However, for the ease of evaluating the correctness of plausible patches, we maintain a dictionary that stores the mapping between the renamed identifiers and their original names. For each vulnerability, we also write a patched program for its code structure transformed version, providing a reference for future dataset users.
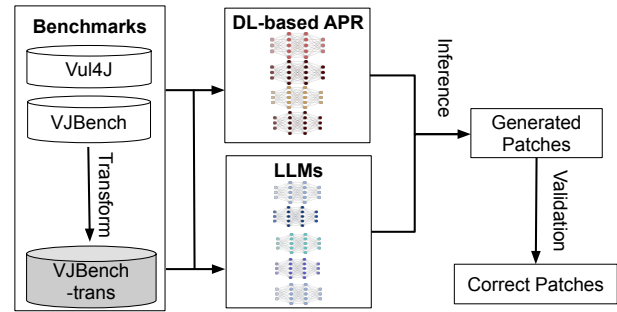


Figure 6: Overview of our study

## 5 EXPERIMENT SETUP

Figure 6 provides an overview of our study. First, we build a new dataset of vulnerabilities, VJBench, that contains 42 new vulnerabilities. We use this new dataset and the original dataset (Vul4J) to benchmark the vulnerability-fixing capabilities of DL-based APR techniques, LLMs and fine-tuned LLMs. Each language model generates 10 patches for each bug through inference. For each APR model, we use its default beam search size and validate its top 10 patches. The generated patches are then validated using test cases and manual verification of all the patches that pass the test cases. Then, we apply code transformations on Vul4J and VJBench to generate VJBench-trans. Finally, we evaluate the impact of code transformations on the vulnerability-repair capabilities of all the LLMs, fine-tuned LLMs and APR techniques.

### 5.1 Dataset

In this work, we focus on fixing single-hunk Java vulnerabilities as state-of-the-art DL-based APR models are designed to fix single-hunk bugs. We filter and obtain 35 single-hunk bugs from Vul4J dataset. Along with the 15 single-hunk vulnerabilities from VJBench, we have a total of 50 Java vulnerabilities. We use the perfect fault localization for these Java vulnerabilities, that is, we use the code lines that are modified in the developers' patches as the buggy lines.

### 5.2 Large Language Model Setups

We evaluate each LLM with two input setups: (1) the buggy lines are commented as part of the input and (2) without the buggy lines. We observe that InCoder fixes more vulnerabilites when the input contains buggy line comments, while the other LLMs perform better without buggy lines. We then report the best-performing setup for each model in the rest of this paper. For fine-tuned LLMs, we follow the input format with buggy line comments used in [38] which is described in Table 2.

We configure each model to generate 10 patches for each vulnerability. For CodeT5, CodeGen, PLBART and InCoder, we set their beam search size to 10. For Codex, we set its parameter $n$, the number of candidates to generate, to 10. Considering the inherent randomness of the sampling method adopted by Codex, we run it twenty-five times for each vulnerability to obtain the average results. We run twenty-five times to control the margin of error small ($\leq 0.3$) at 95% confidence level . We set the sampling temperature of Codex to 0.6, which is shown to have the best performance when sampling ten candidates in prior work [17]. We set the max number

of newly generated tokens to 400 for Codex due to its request rate limit, and to 512 for all other LLMs.

## 5.3 Patch Validation

Codex insertion mode generates code to be inserted between the prefix prompt and the suffix prompt. Since we use the code before and including the buggy line comment as its prefix prompt and the code after the buggy line comment as its suffix prompt, we replace the original buggy code with the code that Codex generates. Similarly, CodeT5 generates code to replace the masked label in its input. PLBART generates the entire patched function that replaces the whole buggy function. CodeGen and InCoder are completion models that generate code to complete the given prefix prompt. We take the first complete function CodeGen and InCoder generate to replace the original buggy function. For all the fine-tuned LLMs, the fine-tuned CodeT5, CodeGen, PLBART and InCoder directly generate the patched code to replace the buggy code.

For each LLM and APR techniques, we first validate the top-10 patches they generate using the test cases from the project. Following prior work [40, 47, 75, 76], *plausible patches* are patches that pass all test cases, while *correct patches* are semantically equivalent to developer patches, and *over-fitted patches* are patches that pass all test cases but are incorrect. We manually inspect each plausible patch to identify if it is a correct patch.

## 6 RESULTS AND FINDINGS

We evaluate the vulnerability fixing capabilities of five LLMs, four fine-tuned LLMs and four DL-based APR techniques on two real-world Java vulnerability benchmarks.

## 6.1 RQ1: Vulnerability Fixing Capabilities

We run Codex twenty-five times and report the average number of fixed vulnerabilities with the margin of error, because Codex's patch generation is non-deterministic. For other LLMs, we only run them once since their patch generation is deterministic (Section 5).

Table 4 shows the fixing capabilities, i.e., the number of vulnerabilities that each approach fixes correctly, of five LLMs, four fine-tuned LLMs and four APR models. We consider the top ten patches since a recent study shows that almost all developers are only willing to examine ten patches at most [57]. Results in Table 4 are reported as X/Y, where X is the number of vulnerabilities correctly fixed by each technique and Y is the number of vulnerabilities that are plausibly fixed. A vulnerability is plausibly fixed by a model if the model generates a plausible patch (definition in Section 5.3).

*6.1.1 LLMs vs. APR Techniques.* We first compare using LLMs as is with APR techniques. Here, *LLMs as is* refers to that we apply Codex and LLMs under zero-shot learning and without fine-tuning. Our results show that Codex exhibits the best fixing capability. Out of a total of 50 vulnerabilities in Vul4J and VJBench, Codex fixes an average of 10.2 vulnerabilities with a margin of error of 0.3 (at 95% confidence). InCoder demonstrates the second best capability, fixing 5 vulnerabilities. The other LLMs and DL-based APR techniques only fix very few vulnerabilities. Overall, LLMs and APR techniques show very limited vulnerability fixing capabilities.

Our finding of Codex performing the best on fixing Java vulnerabilities is consistent with Codex's superior performance in repairing general bugs [74] and in other domains [1, 17, 27, 58], possibly due to its significantly larger model size and training data size as indicated in Table 3. Our result is also consistent with recent work [38] in showing that LLMs without fine-tuning have competitive fixing capabilities – InCoder fix three more vulnerabilities than the best APR technique (RewardRepair). However, while [38] shows that CodeGen, PLBART and InCoder as is can fix 18%-23% general bug of Java APR benchmarks, our result shows that they can fix only 4%(2/50)-10%(5/50) vulnerabilities of Vul4J and VJBench. In real-world, only about 1~7% of bugs are vulnerabilities, resulting in few data for models to learn from. This means that, for neural networks, fixing vulnerabilities is more difficult than general bugs and requires more domain-specific knowledge.

> **Finding 1**: Existing large language models and APR techniques fix very few Java vulnerabilities. Codex fixes 10.2 (20.4%) vulnerabilities on average, exhibiting the best fixing capability.

*6.1.2 LLMs Fine-Tuned with APR Data.* We applied LLMs fined-tuned with general APR data by [38] on the vulnerability benchmarks. We cannot fine-tune Codex as OpenAI does not provide a public API for fine-tuning. Table 4 shows that all the fined-tuned LLMs fix more vulnerabilities than their original models. In detail, fine-tuned InCoder fixes 9 vulnerabilities, 4 more than its original model. The second best models is fine-tuned CodeGen, which fixes 8 vulnerabilities, 6 more than its original model. Fine-tuned CodeT5 and fine-tuned PLBART each fixes 3 and 2 more vulnerabilities.

Overall, fine-tuning with general APR data can improve the fixing capabilities of LLMs for vulnerabilities. First, fine-tuning could adapt LLMs to APR tasks better, making LLMs be aware of generating patches instead of open-ending code or text. Second, though vulnerabilities have special characteristics (root causes) compared to general bugs, some vulnerabilities still share similar repair patterns with general bugs, such as replacing a function argument with another variable, which can be well learned during fine-tuning. Given the scarcity of real-world vulnerability data, our results implicate that fine-tuning LLMs with general APR data can be beneficial.

> **Finding 2**: Fine-tuning with general APR data improves all four LLMs' vulnerability-fixing capabilities. Fine-tuned InCoder fixes 9 vulnerabilities, exhibiting competitive fixing capability compared to Codex's.

We also evaluate the compilation rates (i.e., portions of generated patches that compile) to study the quality of the patches. Uncompilable patches cannot be correct patches. Codex, the best model overall, has a compilation rate of 79.7%, which is significantly higher than that of the best fine-tuned LLM, fine-tuned InCoder (55.2%) and the best APR model, Recoder (57.6%). Fine-tuning notably improves CodeT5 and CodeGen's compilation rates, from 6.4% to 46.8% and from 35.8% to 47.2% respectively. On the other hand, the compilation rate of fine-tuned PLBART is 45.2%, slightly lower than the original PLBART's compilation rate of 47.8%. Despite the higher 65.2% compilation rate of InCoder compared to its fine-tuned model, it generates 82.0% duplicate patches, whereas the fine-tuned InCoder generates patches with more diverse modifications that result in more correct fixes. Overall, compared with compilation rates of repairing general bugs [38], these compilation rates of fixing

Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah

**Table 4: Comparison of LLMs and APR models on fixing Java vulnerabilities. For x/y in a cell, x denotes the number of correctly-fixed bugs, and y is plausibly-fixed bugs (with at least one patch that passes the test cases). RewardR is RewardRepair.**

| | LLMs | | | | | Fine-Tuned LLMs | | | | APR models | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Codex | CodeT5 | CodeGen | PLBART | InCoder | CodeT5 | CodeGen | PLBART | InCoder | CURE | Recoder | RewardR | KNOD |
| VJBench (15) | **4.0**/ 4.6 | 0/0 | 1/2 | 2/3 | 2/2 | 3/4 | 3/4 | 2/3 | 3/4 | 0/1 | 1/2 | 2/3 | 0/0 |
| Vul4J (35) | **6.2**/ 10.9 | 2/2 | 1/6 | 0/4 | 3/4 | 2/7 | 5/8 | 2/6 | 6/9 | 1/4 | 0/4 | 0/2 | 1/1 |
| **Total (50)** | **10.2**/ 15.5 | 2/2 | 2/8 | 2/7 | 5/6 | 5/11 | 8/12 | 4/9 | 9/13 | 1/5 | 1/6 | 2/5 | 1/1 |
| Compilation Rate (%) | 79.7 | 6.4 | 35.8 | 47.8 | 65.2 | 46.8 | 47.2 | 45.2 | 55.2 | 24.5 | 57.6 | 37.7 | 37.3 |

```
private int extend(int v, final int t) {
    int vt = (1 << (t - 1));
-   while(v < vt) {
+   if (v < vt) {                               Developer patch
+   while (v < vt && vt > 0) { t--;             Codex's patch
+   while (!v.equals(vt)) {                     RewardR's patch
}
```

**(a) Vul4J-12 and its uncompilable patches**

```
parser.parseArray(
-       componentClass, array, fieldName);
+       componentType, array, fieldName);          Developer patch
+       componentClass, array, fieldName, null);   LLMs' patch
```

**(b) Vul4J-1 and its uncompilable patches**

**Figure 7: Vul4J-12's and VulJ-1's developer patch and uncompilable patch**

vulnerability are lower. PLBART, CodeGen and InCoder without fine-tuning when repairing general bugs show an average of 65%–73% compilation rate [38], outperforming both of their original and fine-tuned models when repairing vulnerabilities.

Figure 7a shows an example of uncompilable patches of Vul4J-12: The function signature declares t to be final, thus t's value is not allowed to be changed. However, Codex fails to capture this constraint, even though the function signature is only two lines above the buggy line. As a result, it generates code t-- to decrease t's value which makes the patch uncompilable. Similary, RewardR ignores the fact that v and vt are both of type int, and invokes the invalid function equals on them. Figure 7b shows another example of uncomplable patch for Vul4J-1: parseArray is a method defined in another class in the project that accepts two or three arguments only. All the four fine-tuned LLMs generate the same uncompilable patches where they pass null as the fourth argument, because they do not have the information that parseArray does not accept four arguments.

These results suggest that LLMs' abilities to learn code syntax could be improved. Recent work [40, 76] are steps in the right direction to add domain knowledge to models to help them learn code syntax and semantics. Another direction is prompt engineering, such as providing method signatures or type information in the prompt to specify the constraints. This would enable LLMs to utilize syntax information from across the entire project, rather than being limited to the code within the buggy function.

> **Finding 3**: Codex has the highest compilation rate of 79.7%. Other LLMs (fine-tuned or not) and APR techniques have low compilation rates (the lowest of 6.4% with CodeT5 and the rest between 24.5% to 65.2%), showing a lack of syntax domain knowledge.

```
xmlIn = XMLInputFactory.newInstance();
xmlIn.setProperty(XMLInputFactory.IS_SUPPORTING_
    EXTERNAL_ENTITIES, Boolean.FALSE);
+ xmlIn.setproperty(XMLInputFactory.SUPPORT_DTD,
+                   Boolean.FALSE);
```

**(a) Vul4J-47 and its developer patch**

```
xmlIn = XMLInputFactory.newInstance();
- xmlIn.setProperty(XMLInputFactory.IS_SUPPORTING_
    EXTERNAL_ENTITIES, Boolean.FALSE);
+ xmlIn.setproperty(XMLInputFactory.IS_SUPPORTING_
    EXTERNAL_ENTITIES, Boolean.TRUE);       Fine-tuned CodeGen's patch
```

**(b) Vul4J-47 and the incorrect patch generated by fine-tuned CodeGen**

**Figure 8: Java vulnerability Vul4J-47 and its patches**

## 6.2 RQ2: What kinds of vulnerabilities do LLMs and learning-based APR techniques fix?

Table 5 shows the vulnerabilities that are correctly fixed by the LLMs, fine-tuned LLMs, and APR techniques. In total, 16 vulnerabilities (belonging to ten CWE categories as shown in column *CWE* with their description in column *Description*) from both benchmarks are fixed by at least one of the models. The IDs of these vulnerabilities are listed under column *Vul. ID*. Some vulnerabilities belong to no specific CWE category and are listed as *unk*.

Vul4J-47 is a vulnerability that only Codex can fix. Figure 8a shows the developer patch for Vul4J-47 of type CWE-611 (Improper Restriction of XML External Entity Reference) and CWE-918 (Server-Side Request Forgery). The correct fix requires inserting a statement xmlIn.setProperty(XMLInputFactory. SUPPORT_DTD, Boolean.FALSE) to disable the support of Document Type Definition (DTD), because DTD can be used to perform server-side request forgery (SSRF) attacks. The original buggy code only disables the support for external entities by setting the IS_SUPPORTING_EXTERNAL_ ENTITIES property to false, which is not enough to prevent the attack. Figure 8b shows an incorrect patch generated by fine-tuned CodeGen, which replaces the Boolean.FALSE with Boolean.TRUE. In general, except Codex, other LLMs and fine-tuned LLMs only fix vulnerabilities that require simple modifications such as deleting statements or replacing variable/method names.

On the other hand, Codex fixes 15 out of the 16 vulnerabilities (the union of all bugs, for which Codex generates at least one correct patch in twenty-five runs). The one vulnerability fixed by other LLMs but not Codex is Vul4J-39 of type CWE-200 (Exposure of Sensitive Information to an Unauthorized Actor). This vulnerability can be fixed by simply deleting the entire buggy code. However, for Vul4J-39, Codex generates patches by applying different modifications to the buggy code, rather than deleting it.

**Table 5: Detailed description of the vulnerabilities fixed by each LLM, fine-tuned LLM, and DL-based APR technique**

| Vul. ID | CWE | Description | LLMs | | | | | Fine-tuned LLMs | | | | APR Techniques | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Codex | CodeT5 | CodeGen | PLBART | InCoder | CodeT5 | CodeGen | PLBART | InCoder | CURE | Recoder | RewardR | KNOD |
| Vul4J-1 | 20 | Improper Input Validation | ✓ | | | | | | ✓ | | ✓ | | | | |
| Vul4J-4 | unk | / | ✓ | | | | | ✓ | ✓ | | ✓ | | | | |
| Vul4J-5 | unk | / | ✓ | | | | | | | ✓ | ✓ | | | | |
| Vul4J-12 | 835 | Infinite Loop | ✓ | ✓ | | | | | ✓ | ✓ | ✓ | ✓ | | | |
| Vul4J-19 | unk | / | ✓ | | | | | | | | | | | | |
| Vul4J-20 | unk | / | ✓ | | | | | | | | | | | | |
| Vul4J-25 | 39 | Cross-site Scripting | ✓ | | | | | | | | | | | | |
| Vul4J-39 | 200 | Sensitive Information | | ✓ | ✓ | | ✓ | ✓ | ✓ | | ✓ | | | | |
| Vul4J-47 | 611 | Improper External References | ✓ | | | | | | | | | | | | |
| Vul4J-50 | 79 | Cross-site Scripting | ✓ | | | | ✓ | | ✓ | | ✓ | | | | |
| Vul4J-59 | 79 | Cross-site Scripting | ✓ | | | | ✓ | | | | | | | | |
| Vul4J-73 | 522 | Protected Credentials | ✓ | | | | | | | | | | | | ✓ |
| Halo-1 | 22 | Path Traversal | ✓ | | | | ✓ | | ✓ | | ✓ | | | | |
| Jenkins-2 | 200 | Sensitive Information | ✓ | | | ✓ | ✓ | ✓ | | | ✓ | | | ✓ | |
| Jenkins-3 | 200 | Sensitive Information | ✓ | | ✓ | ✓ | | ✓ | ✓ | ✓ | | | | | |
| Ratpack-1 | 74 | Improper Neutralization | ✓ | | | | | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ | |
| #Total: 16 | | | 15 | 2 | 2 | 2 | 5 | 5 | 8 | 4 | 9 | 1 | 1 | 2 | 1 |

```
- HttpHeaders nettyHeaders = new DefaultHttpHeaders(false);
+ HttpHeaders nettyHeaders = new DefaultHttpHeaders();
```

**Figure 9: Java vulnerability Ratpack-1 and its developer patch**

Ratpack-1, Vul4J-12, Vul4J-39 and Jenkins-2 are four vulnerabilities fixed by the most number (6-7 out of 13) of models. Ratpack-1 (Figure 9) when initializing `DefaultHttpHeaders`, sets the constructor argument to `false`, which disables the validation for user-supplied header values. The correct patch is simply removing `false` or changing it to `true` to enable the validation. The fix for Vul4J-12 (Figure 7a) is to change the keyword `while` to `if`, and the fix for both Jenkins-2 and Vul4J-39 is to simply delete of an `if` statement that exposes sensitive information to unauthorized actors. The simplicity of these patches are evident from the number of models that can fix them.

> **Finding 4**: Large language models and APR techniques, except Codex, only fix vulnerabilities that require simple changes, such as deleting statements or replacing variable/method names.

Surprisingly, the nine LLMs and four APR techniques fix none of the six new CWE types that VJBench adds, which shows that our VJBench helps reveal the limitations of existing LLMs and APR techniques in fixing Java vulnerabilities. This calls for new techniques that can fix CWE-172, CWE-325, CWE-347, CWE-444, CWE-668, and CWE-1295. In addition, for CWE-611 that is covered by Vul4J's Vul4J-47, we add two instances of this CWE type (Quartz-1 and Retrofit-1) in VJBench. Codex fixes Vul4J-47, but none of the LLMs and APR techniques fixes the additional Quartz-1 and Retrofit-1. This shows that VJBench complements Vul4J even on CWE categories that Vul4J has already covered.

Figure 10 shows Retrofit-1 of CWE-611 category. None of the models fixes Retrofit-1. The correct patch is to prevent XML External Entity attacks by calling `xmlInputFactory.setProperty(...)` to disable the support for external entities and DTD. But as LLMs are not provided with information that the vulnerability is about XML External Entity attacks (as suggested by the CWE type), they only make changes on the buggy code (Figure 10b) unrelated to XML properties. Figure 11 shows Jenkins-1 of CWE-325 (Missing cryptographic step), a new CWE category that VJBench adds. The correct fix for the bug is adding if-condition to check the permission

```
  this.type = type;
+ xmlInputFactory.setProperty(XMLInputFactory.IS_SUPPORTING_
+     EXTERNAL_ENTITIES, false);
+ xmlInputFactory.setProperty(XMLInputFactory.SUPPORT_DTD,
+                     false);
```

**(a) Developer patch of Retrofit-1**

```
- this.type = type;
+ this.type = (Class<T>) type;
```

**(b) Incorrect fix generated by InCoder**

**Figure 10: Java vulnerability Retrofit-1 and its patches.**

```
- for (NodeMonitor monitor : NodeMonitor.getAll())
-     r.put(monitor.getClass().getName(), monitor.data(this));
+ if (hasPermission(CONNECT)) {
+     for (NodeMonitor monitor : NodeMonitor.getAll())
+         r.put(monitor.getClass().getName(), monitor.data(this));
+ }                                              Developer patch
+ for (NodeMonitor monitor : NodeMonitor.getAll())
+     r.put(monitor.data(this));                 Codex's patch
```

**Figure 11: Java vulnerability Jenkins-1 and its patches**

before the for-loop to restrict the access to `NodeMonitor`. As Codex's patch shown in Figure 11, all the models fail to fix the bug because they only apply general modifications to the `for`-loop and are unaware that the bug is related to the permission restriction. Further, the `hasPermission` method and the `CONNECT` variable are declared outside of the buggy function, thus the models have no knowledge about their usages. This reflects two problems for LLMs to fix Java vulnerabilities: (1) With only buggy lines pointed out, LLMs fail to generate patches targeting the vulnerability. This suggests that it is necessary to provide LLMs with more information about the vulnerability, such as CWE types. (2) More project-specific information is needed for LLMs to fix vulnerabilities, i.e., providing LLMs with related methods and variables declared outside of the buggy function.

Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah

**Table 6: Impact of code transformation on LLMs' and APR models' vulnerability repair capabilities. For Codex, x ± y: x denotes the average number of correctly fixed bug, and y denotes the margin of error (95% confidence).**

| | LLMs | | | | | Fine-Tuned LLMs | | | | APR Techniques | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Codex | CodeT5 | CodeGen | PLBART | InCoder | CodeT5 | CodeGen | PLBART | InCoder | CURE | Recoder | RewardR | KNOD |
| No transformation | 10.2 ±0.3 | 2 | 2 | 2 | **5** | **5** | **8** | **4** | **9** | 1 | 1 | 2 | 1 |
| Rename only | 8.1 ±0.3 | 0 | 1 | 0 | **2** | 4 | 6 | 1 | 7 | 0 | 1 | 1 | 1 |
| Code structure change only | 10.0 ±0.3 | 0 | 2 | 2 | **1** | 4 | 6 | 4 | 7 | 0 | 1 | 1 | 2 |
| Rename + code structure change | 8.7 ±0.4 | 0 | 1 | 1 | **1** | 3 | 4 | 3 | 4 | 0 | 1 | 1 | 0 |

> **Finding 5**: Our new VJBench benchmark reveals that large language models and APR techniques fail to fix many CWE types, including CWE-172 (Encoding error), CWE-325 (Missing cryptographic step), CWE-444 (HTTP request smuggling), CWE-668 (Exposure of resource to wrong sphere), and CWE-1295 (Debug messages revealing unnecessary information).

## 6.3 RQ3: Fixing Capabilities on Transformed Vulnerabilities

To mitigate the training-testing data overlapping threat, we apply code transformations to the benchmarks to study the generalization abilities of Codex and LLMs on unseen data (Section 4). Table 6 shows the number of vulnerabilities that LLMs as is, fine-tuned LLMs, and APR techniques can fix in four settings: (1) *No transformation*—the original vulnerability dataset, (2) *Rename only*—only identifier renaming is applied, (3) *Code structure change only*—only code structure change is applied, and (4) *Rename + code structure change*—both transformations are applied.

Overall, code transformations make LLMs (fine-tuned or not) and APR techniques fix fewer vulnerabilities. For example, fine-tuned InCoder fixes nine vulnerabilities in Vul4J and VJBench (no transformation), but only fixes four fully transformed vulnerabilities (Rename + Code structure change). The impact of transformation is smaller on some models, e.g., Codex and fine-tuned CodeT5, demonstrating these models' robustness against code transformations and generalized learning capabilities. This result, to some extent, addresses the threat of Codex's non-public training data and reveals Codex's strong learning and vulnerability-fixing capability. Many models only fix two or fewer vulnerabilities without transformations, thus the impact of transformations cannot be big for these models. However, we see a general trend across almost all models that these code transformations make models fix fewer number of vulnerabilities.

Figure 12a shows an example, Halo-1, whose correct fix is to call `normalize()` on `pathToCheck` to remove any redundant elements in the file path. This bug can be correctly fixed by Codex, fine-tuned CodeGen, and fine-tuned InCoder. Yet, after applying both transformations, only Codex can fix it (Figure 12b).

Different transformations have different effects but each transformation significantly affects at least one LLM. For example, although identifier renaming has small effect on fine-tuned CodeT5, it decreases the number of vulnerabilities that fine-tuned PLBART fixes by three. The result shows that our code transformation effectively tests the generalization ability of LLMs on unseen data.

One interesting observation is that some models fix transformed vulnerabilities that they cannot fix in the original dataset. This is a reasonable phenomenon because our transformation may convert a code snippet into a simpler form for the models to fix. For example,

```
public static void checkDirectoryTraversal(...) { ...
-     if (pathToCheck.startsWith(parentPath.normalize())) {
+     if (pathToCheck.normalize().startsWith(parentPath)) {
```

(a) Halo's original buggy code and its correct patch

```
public static void examinePathManipulation(...) { ...
-   Path normalizedBasePath = basePath.normalize();
-   if (!examinePath.startsWith(normalizedBasePath)) {
+   Path normalizedExaminePath = examinePath.normalize();
+   if (!normalizedExaminePath.startsWith(basePath)) {
```

(b) Halo's buggy code after Rename+Code structure change and its correct patch generated by Codex

**Figure 12: Halo-1 before and after transformation**

```
- if (... || !Pattern.compile(getUrlRegex(), ...).matcher(
-     String.valueOf(value)).matches()) {
+ if (... || !Pattern.compile(getUrlRegex(), ...).matcher(
+     String.valueOf(value).trim()).matches()) {
```

(a) VUL4J-30's original buggy line and its correct patch

```
  String urlRegex = getUrlRegex();
  Pattern p = Pattern.compile(urlRegex, ...);
- String s = String.valueOf(value);
+ String s = String.valueOf(value).trim();
  Matcher m = p.matcher(s);
  if (... || m.matches()) {
```

(b) VUL4J-30's buggy line after code transformation and its correct patch generated by fine-tuned LLMs.

**Figure 13: Vul4J-30 before and after code structure change**

Vul4J-30 is a bug that none of the models fixes in its original form, but its transformed version is fixed by all four fine-tuned LLMs when code structure transformation is applied. Figure 13 shows that the fix of Vul4J-30 is to call `trim()` on `String.valueOf(value)`. The original vulnerability is hard to fix as `String.valueOf(value)` is a part of a complex if-condition. Yet, after code transformation, `String.valueOf(value)` stands out as a single statement, which is easier for LLMs to repair. This phenomenon suggests that equivalent code transformation could be a promising direction to simplify the vulnerable code and enhance the effectiveness of fixing vulnerabilities.

> **Finding 6**: Code transformations make large language models and APR techniques fix fewer number of vulnerabilities. Some models such as Codex and fine-tuned CodeT5 are more robust to code transformations. On the other hand, some transformations make vulnerabilities easier to fix.

## 7 THREATS TO VALIDITY

Java vulnerabilities are diverse. It is hard for benchmarks to represent all of them. Thus, our findings might not generalize to all Java vulnerabilities. We address this threat by expanding the existing Java vulnerability benchmark with a new dataset of vulnerabilities.

We rely on developers' patches to assess whether a vulnerability is fixed. Developers may make a mistake in fixing vulnerabilities. Therefore, our ground truth might be incorrect. We mitigate this threat by only looking at vulnerabilities that are publicly disclosed in the NVD dataset that are reproducible and include test cases indicating that the fixed version is no more exploitable.

Another threat is that Codex (and other LLMs) may have been trained on the vulnerability patches in Vul4J and VJBench dataset. To mitigate this problem, we apply code transformations to create semantically equivalent vulnerabilities that are not included in their training dataset. Then we apply Codex to repair these transformed programs to prove that Codex is indeed able to repair new vulnerabilities that it has not seen.

## 8 RELATED WORK

### 8.1 DL-Based Vulnerability Fixing Techniques

Much work uses DL to fix vulnerabilities. Encoder-decoder approaches have been proposed for repairing C vulnerabilities: [29] fine-tuned a CodeT5 model with C vulnerability repair data; [19] trained a transformer model on a large bug fixing dataset and then tuned on a small vulnerability fixing dataset, but they use sequence accuracy as the evaluation metric rather than practical APR settings. Previous work [35] applied both CodeBERT and GraphCodeBert to fix vulnerabilities, but they only evaluated on a *synthetic* vulnerability database, the Juliet 1.1 C/C++ test suite [14], which is a benchmark for evaluating static analyzers only. As a result, the vulnerabilities in the dataset are isolated and simplified to fit within a few lines and are not representative of code vulnerabilities in the production. Our work is different since we use a dataset of *real-world* vulnerabilities for our evaluation, making our results closer to what researchers and developers can expect of the quality of LLM vulnerability repair in real-world production code.

Prior work [58] applied LLMs with zero-shot learning to repair seven hand-crafted C/Python vulnerabilities and 12 real-world C vulnerabilities. They explored the effectiveness of different prompt templates and used the static analysis tool CodeQL or C sanitizers to detect the vulnerabilities to incorporate the obtained error messages into the input prompts. Our work differs from [58] in several main aspects. First, we study not only LLMs but also DL-based APR tools and LLMs fine-tuned with general APR data. Second, we evaluate our approach on a larger dataset of 50 real-world Java vulnerabilities. Third, we apply code transformations to mitigate the data leakage problem and suggest a new direction of using transformations to simplify the repair for some vulnerabilities. Most vulnerabilities in Vul4J and VJBench cannot be detected by state-of-the-art Java security analysis tools, so we cannot incorporate error messages in the input prompts as [58] did.

### 8.2 Vulnerability Benchmarks

Previous work proposed benchmarks and datasets to help evaluate vulnerability fixing approaches. Maestro [61] propose a platform for benchmarking tools on Java and C++ vulnerabilities. As Maestro does not support running LLMs and APR models, we directly use the same Java vulnerability dataset, Vul4J [16], with our new dataset VJBench. Other benchmarks and datasets of real-world vulnerabilities have been proposed [12, 24, 56, 62]. However, these datasets only contain code snippets from the fixing commits and do not have test cases. Therefore, such datasets can only support code matching when evaluating the correctness of patches, and cannot be used in automated program repair in practice.

### 8.3 LLMs for Repair and Other Tasks

Researchers use LLMs to improve many software engineering tasks such as automated program repair [40, 50, 63], auto-complete suggestions [25], and pair-programming [37]. Much work also discusses the implication of LLMs for software developers [23, 27, 51] and current limitations of LLMs [9, 20, 68]. Our work explores a different application domain of LLMs, with its own challenges (vulnerabilities are notoriously difficult to fix [52]) that have not been well explored yet.

## 9 CONCLUSION

This work is the first to investigate LLMs' and DL-based APR models' capacity at repairing vulnerabilities in Java. We evaluate five LLMs, four fine-tuned LLMs, and four DL-based APR techniques on two real-world Java vulnerability benchmarks including a new one that we create. We use code transformations to address the training and testing data overlapping threat of LLMs and create a new Java vulnerability repair benchmark VJBench, and its transformed version VJBench-trans. We find that existing LLMs and APR models fix very few Java vulnerabilities, and call for new research innovations to improve automated Java vulnerability repair such as creating larger vulnerability repair training datasets, fine-tuning LLMs with such data, exploring few-shot learning, and leveraging simplifying transformations to improve program repair.

**Replication package**: Our benchmark and artifacts are available at [5].

## REFERENCES

[1] 2022. Codex. https://beta.openai.com/docs/guides/code
[2] Accessed: 2022. Guidance for preventing, detecting, and hunting for exploitation of the Log4j 2 vulnerability. https://www.microsoft.com/en-us/security/blog/2021/12/11/guidance-for-preventing-detecting-and-hunting-for-cve-2021-44228-log4j-2-exploitation/.
[3] Accessed: 2023. NLTK Documentation. https://www.nltk.org/howto/wordnet.html.
[4] Accessed: 2023. NVD Data Feeds. https://nvd.nist.gov/vuln/data-feeds.
[5] Accessed: 2023. Replication package of this work. https://github.com/lin-tan/llm-vul.
[6] Accessed: 2023. src2abs GitHub Repository. https://github.com/micheletufano/src2abs.
[7] Accessed: 2023. Understanding the Impact of Apache Log4j Vulnerability. https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html.
[8] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

[9] Owura Asare, Meiyappan Nagappan, and N Asokan. 2022. Is GitHub's Copilot as Bad As Humans at Introducing Vulnerabilities in Code? *arXiv preprint arXiv:2204.04741* (2022).

[10] Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, Alex Rebert, and Ned Williamson. 2018. The mayhem cyber reasoning system. *IEEE Security & Privacy* 16, 2 (2018), 52–60.

[11] Björn Barz and Joachim Denzler. 2020. Do we train on test data? purging cifar of near-duplicates. *Journal of Imaging* 6, 6 (2020), 41.

[12] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39. https://doi.org/10.1145/3475960.3475985

[13] Sid Black, Gao Leo, Phil Wang, Connor Leahy, and Stella Biderman. 2021. GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow. (March 2021). https://doi.org/10.5281/zenodo.5297715

[14] Tim Boland and Paul E Black. 2012. Juliet 1. 1 C/C++ and java test suite. *Computer* 45, 10 (2012), 88–90.

[15] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[16] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 464–468. https://doi.org/10.1145/3524842.3528482

[17] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[18] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1943–1959. https://doi.org/10.1109/TSE.2019.2940179

[19] Zimin Chen, Steve James Kommrusch, and Martin Monperrus. 2022. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering* (2022).

[20] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C Desmarais, Zhen Ming, et al. 2022. GitHub Copilot AI pair programmer: Asset or Liability? *arXiv preprint arXiv:2206.15331* (2022).

[21] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=SJeqs6EFvB

[22] Dawn Drain, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. 2021. DeepDebug: Fixing Python Bugs Using Stack Traces, Backtranslation, and Code Skeletons. *CoRR* abs/2105.09352 (2021). arXiv:2105.09352 https://arxiv.org/abs/2105.09352

[23] Neil A Ernst and Gabriele Bavota. 2022. AI-Driven Development Is Here: Should You Worry? *IEEE Software* 39, 2 (2022), 106–110.

[24] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512. https://doi.org/10.1145/3379597.3387501

[25] Zhiyu Fan, Xiang Gao, Abhik Roychoudhury, and Shin Hwei Tan. 2022. Improving automatically generated code from Codex via Automated Program Repair. *arXiv preprint arXiv:2205.10583* (2022).

[26] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. Code-BERT: A Pre-Trained Model for Programming and Natural Languages. *CoRR* abs/2002.08155 (2020). arXiv:2002.08155 https://arxiv.org/abs/2002.08155

[27] James Finnie-Ansley, Paul Denny, Brett A Becker, Andrew Luxton-Reilly, and James Prather. 2022. The robots are coming: Exploring the implications of openai codex on introductory programming. In *Australasian Computing Education Conference*. 10–19.

[28] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. https://doi.org/10.48550/ARXIV.2204.05999

[29] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: A T5-Based Automated Software Vulnerability Repair. (2022).

[30] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe memory-leak fixing for c programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 459–470.

[31] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond tests: Program vulnerability repair via crash constraint extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27.

[32] github. 2022. GitHub. https://github.com/

[33] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. *CoRR* abs/2009.08366 (2020). arXiv:2009.08366 https://arxiv.org/abs/2009.08366

[34] Raphael Hiesgen, Marcin Nawrocki, Thomas C Schmidt, and Matthias Wählisch. 2022. The Race to the Vulnerable: Measuring the Log4j Shell Incident. In *Network Traffic Measurement and Analysis Conference (TMA)*.

[35] Kai Huang, Su Yang, Hongyu Sun, Chengyi Sun, Xuejun Li, and Yuqing Zhang. 2022. Repairing Security Vulnerabilities Using Pre-trained Programming Language Models. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 111–116. https://doi.org/10.1109/DSN-W54100.2022.00027

[36] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 539–554.

[37] Saki Imai. 2022. Is GitHub Copilot a Substitute for Human Pair-programming? An Empirical Study. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 319–321.

[38] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020*.

[39] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. In *Proceedings of the International Conference on Software Engineering*. https://doi.org/10.1109/ICSE48619.2023.00111

[40] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. Cure: Code-aware neural machine translation for automatic program repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1161–1173.

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

[42] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. Memfix: static analysis-based repair of memory deallocation errors for c. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 95–106.

[43] Xiaodan Li, Xiaolin Chang, John A Board, and Kishor S Trivedi. 2017. A novel approach for software vulnerability classification. In *2017 Annual Reliability and Maintainability Symposium (RAMS)*. IEEE, 1–7.

[44] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*. 55–56.

[45] Xi Victoria Lin, Todor Mihaylov, Mikel Artetxe, Tianlu Wang, Shuohui Chen, Daniel Simig, Myle Ott, Naman Goyal, Shruti Bhosale, Jingfei Du, Ramakanth Pasunuru, Sam Shleifer, Punit Singh Koura, Vishrav Chaudhary, Brian O'Horo, Jeff Wang, Luke Zettlemoyer, Zornitsa Kozareva, Mona T. Diab, Veselin Stoyanov, and Xian Li. 2021. Few-shot Learning with Multilingual Language Models. *CoRR* abs/2112.10668 (2021). arXiv:2112.10668 https://arxiv.org/abs/2112.10668

[46] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. 2007. AutoPaG: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. 329–340.

[47] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair. In *ISSTA* (Virtual Event, USA). ACM, 101–114. https://doi.org/10.1145/3395363.3397369

[48] Siqi Ma, David Lo, Teng Li, and Robert H Deng. 2016. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. 711–722.

[49] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 468–478.

[50] Ehsan Mashhadi and Hadi Hemmati. 2021. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 505–509.

[51] Ekaterina A Moroz, Vladimir O Grizkevich, and Igor M Novozhilov. 2022. The Potential of Artificial Intelligence as a Method of Software Developer's Productivity Improvement. In *2022 Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*. IEEE, 386–390.

[52] Patrick J Morrison, Rahul Pandita, Xusheng Xiao, Ram Chillarege, and Laurie Williams. 2018. Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering* 23, 3 (2018), 1383–1421. https://doi.org/10.1145/3180155.3182553

[53] Paul Muntean, Martin Monperrus, Hao Sun, Jens Grossklags, and Claudia Eckert. 2019. Intrepair: Informed repairing of integer overflows. *IEEE Transactions on Software Engineering* 47, 10 (2019), 2225–2241.

[54] David J Musliner, SE Friedman, M Boldt, J Benton, M Schuchard, P Keller, and S McCamant. 2015. Fuzzbomb: Autonomous cyber vulnerability detection and repair. In *Fourth International Conference on Communications, Computation, Networks and Technologies*.

[55] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A conversational paradigm for program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[56] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: a cross-language vulnerability dataset with commit data. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1565–1569. https://doi.org/10.1145/3468264.3473122

[57] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering*.

[58] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2022. Examining Zero-Shot Vulnerability Repair with Large Language Models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 1–18.

[59] José D'Abruzzo Pereira, Naghmeh Ivaki, and Marco Vieira. 2021. Characterizing Buffer Overflow Vulnerabilities in Large C/C++ Projects. *IEEE Access* 9 (2021), 142879–142892. https://doi.org/10.1109/ACCESS.2021.3120349

[60] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. 2009. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 87–102.

[61] Eduard Pinconschi, Quang-Cuong Bui, Rui Abreu, Pedro Adão, and Riccardo Scandariato. 2022. Maestro: a platform for benchmarking automatic program repair tools on software vulnerabilities. In *International Symposium on Software Testing and Analysis*. 789–792.

[62] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 383–387.

[63] Julian Aron Prenner, Hlib Babii, and Romain Robbes. 2022. Can OpenAI's Codex Fix Bugs?: An evaluation on QuixBugs. In *2022 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 69–75.

[64] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[65] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *CoRR* abs/1910.10683 (2019). arXiv:1910.10683 http://arxiv.org/abs/1910.10683

[66] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. 2018. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th international conference on mining software repositories*. 10–13.

[67] Stelios Sidiroglou and Angelos D Keromytis. 2005. Countering network worms through automatic patch generation. *IEEE Security & Privacy* 3, 6 (2005), 41–49.

[68] Adam Sobieszek and Tadeusz Price. 2022. Playing Games with Ais: The Limits of GPT-3 and Similar Large Language Models. *Minds and Machines* 32, 2 (2022), 341–364.

[69] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 99–108.

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. *CoRR* abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762

[71] Ben Wang and Aran Komatsuzaki. 2021. GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model. (May 2021).

[72] Tielei Wang, Chengyu Song, and Wenke Lee. 2014. Diagnosis and emergency patch generation for integer overflow exploits. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 255–275.

[73] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. 8696–8708.

[74] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2022. Practical Program Repair in the Era of Large Pre-trained Language Models. *arXiv preprint arXiv:2210.14179* (2022).

[75] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 1506–1518.

[76] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A syntax-guided edit decoder for neural program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 341–353.