ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software

Xiang Mei
Ira A. Fulton School of Engineering
Arizona State University
Email: xmei5@asu.edu

Jordi Del Castillo
Tandon School of Engineering
New York University
Email: jordi.d@nyu.edu

Haoran Xi
Tandon School of Engineering
New York University
Email: hx759@nyu.edu

Hammond Pearce
Computer Science and Engineering
University of New South Wales
Email: hammond.pearce@unsw.edu.au

Brendan Dolan-Gavitt
Tandon School of Engineering
New York University
Email: brendandg@nyu.edu

***Abstract*—Recent advances in program repair and large language models (LLMs) have shown promise for *automatic* repair of security vulnerabilities in code. However, there is a shortage of large-scale benchmarks and datasets that can be used to evaluate and improve such techniques: existing datasets have a relatively small number of vulnerabilities (e.g., only 30 in ExtractFix) and lack triggering inputs that demonstrate the vulnerabilities. They are also static, meaning that (particularly with LLM-based techniques, which can inadvertently include the patched version of the code in their training data) techniques may overfit to the benchmark. In this paper, we present ARVO: an Atlas of Reproducible Vulnerabilities in Open source software. By sourcing vulnerabilities from C/C++ projects discovered by Google's OSS-Fuzz, we are able to automatically build a benchmark for vulnerability repair with nearly 3,000 vulnerabilities across more than 200 projects, each with a triggering input and the canonical developer-written patch for fixing the vulnerability. Moreover, our dataset can be automatically updated as OSS-Fuzz finds new vulnerabilities, mitigating issues of training data contamination and overfitting. To demonstrate the potential for this dataset, we provide an initial baseline evaluation of current language models like GPT-4 and WizardCoder for vulnerability repair. We find that the best model achieves only a 37% success rate, indicating substantial room for improvements in this area.**

1. Introduction

Vulnerabilities in software are both common and damaging: in 2021 alone, more than 20,000 vulnerabilities were tracked by the National Vulnerability Database (NVD), 4,073 of which were classified as high severity. Reducing the manual effort needed to fix such vulnerabilities would allow developers to deploy patches more quickly and reduce the window of opportunity in which attackers can exploit them. Recently, large language models (LLMs) have shown surprising capabilities for this task [1], [2], [3], [4], but evaluations of such techniques on real-world vulnerabilities are scarce, in large part because large datasets of vulnerabilities suitable for evaluating repair techniques are themselves in short supply.

In this paper we aim to remedy this shortage of benchmark data by introducing a new dataset, derived from Google's OSS-Fuzz project, that is suitable for evaluating and improving vulnerability repair techniques. We focus primarily on C/C++ projects due to their popularity and the potential severity of bugs in these languages. Our benchmark has the following key features: 1) It has a large number of reproducible vulnerabilities (3,479 vulnerabilities across 231 projects at the time of this writing); 2) Each vulnerability comes with a proof-of-concept 'triggering' input that can be used to test for the presence of the vulnerability; 3) For most vulnerabilities (2,908) we are able to isolate and provide the precise developer patch that fixes the vulnerability; and 4) We provide infrastructure for automatically applying a proposed fix to the code, rebuilding it, and testing whether the vulnerability is still present. These features, as well as the fact that it can be continuously updated with new vulnerabilities without manual effort, make it ideal for systematic evaluations of vulnerability repair. It also provides a robust basis for creating benchmarks that evaluate of related tasks, such as vulnerability discovery and automated exploit generation.

Our dataset is constructed by scraping Google's OSS-Fuzz bug tracker [5], which reports findings from continuous fuzzing over 300 open-source projects. However, the reports are by themselves not enough to reproduce each vulnerability: while they provide the revision and triggering input found by the fuzzer, reliably building the target software and its dependencies at the vulnerable version, as well as locating the precise patch that fixes the issue, require additional effort. The raw issue tracker is also not particularly user-friendly, requiring significant effort and computational resources to build all the software and test patches. ARVO fills in these missing pieces by automatically finding and building dependencies at the correct versions, validating that the vulnerability can be reproduced, creating a pre-built Docker image for each vulnerability, and providing an automated infrastructure that allows benchmark consumers get an evaluation score by simply providing the patched code in JSON format. We discuss the design of ARVO and how it accomplishes these tasks in Section 3.

Because we are interested in using this benchmark to help improve the state of the art in LLM-based vulnerability repair, we also conduct initial experiments to validate the usefulness of our dataset and provide initial baselines. We measure: 1) the performance of GPT-3.5 across the full set of "simple" vulnerabilities (where the fix affects only a single function), 2) the performance of four state-of-the-art models (OpenAI's Codex, GPT-3.5, and GPT-4 models, as well as the open-source WizardCoder model) on a randomly chosen subset of 100 vulnerabilities, using two different prompting strategies, and 3) the effect of training data contamination, by comparing the performance of GPT-3.5 and GPT-4 on 100 vulnerabilities found before, and 100 found after, the examined models' training data cutoff (September 2021).

In summary, we make the following contributions:

1) We present a new dataset suitable for evaluating vulnerability repair techniques, which is (to the best of our knowledge) the largest of its kind.
2) We provide an initial evaluation of current LLMs on our benchmark, finding that the best-performing model (GPT-4) is only able to repair 37% of our vulnerabilities and no evidence of training data contamination.
3) We make our dataset construction and evaluation infrastructure open-source, along with 30 TB of Docker images that can be used to reproduce each vulnerability and test repairs.

2. Background

2.1. Fuzzing and OSS-Fuzz

One of the most widespread techniques for finding vulnerabilities in software, particularly software written in unsafe languages such as C/C++, is fuzzing [6], [7]. Since the release of American Fuzzy Lop (AFL) in 2013, fuzzing has attracted considerable attention from academic researchers and industry, and has been used to find vulnerabilities in a wide range of critical software. Fuzzing mutates inputs to a target program, runs the program on those inputs, and selects inputs that expose new *coverage* for further mutation.

Created by Google in 2016, OSS-Fuzz [5] is an open-source project that performs continuous fuzzing to detect and report security vulnerabilities in over 1,000 open-source projects. Each project is expected to provide a *fuzz harness* that specifies API functions in the project to test. OSS-Fuzz then monitors the project repository, builds the software as new commits are made, fuzzes them with a variety of fuzzers and sanitizers (e.g., Address Sanitizer), automatically reports crashes found by the fuzzers, and periodically checks whether the project has fixed the reported vulnerability. It has helped find and fix more than 10,000 vulnerabilities, as of August 2023.

2.2. Vulnerability Repair

Because developer time is limited and vulnerabilities are plentiful, there is a need for techniques that can *automatically* fix vulnerabilities with minimal human intervention. Vulnerability repair is a special case of *automated program repair* (APR) [8], but focuses on fixing bugs with security impact; in this paper, we focus particularly on memory safety vulnerabilities in C/C++ software, both because such vulnerabilities are particularly serious (leading, e.g., to remote code execution) and because most "infrastructure" software (like OpenSSL) is still written in C or C++. Vulnerability repair typically assumes the existence of a *localization oracle* that tells the repair system where the fix should be applied; in ARVO, we use the developer-provided patches for this purpose, under the assumption that code fixed by the patch represents the root cause of the issue. In the absence of a localization oracle (i.e. when fixing a previously unknown vulnerability), automated root cause analysis techniques [9] could be employed instead. A repair is considered successful if the vulnerability is no longer present (often checked by running the target program on a proof-of-concept input that triggers the vulnerability) and repair does not change unrelated program functionality (as tested, e.g., by running the project test suite).

Automatically validating a proposed vulnerability repair is challenging. Even if a proof-of-concept input no longer triggers the vulnerability in the patched code, the underlying flaw may still be present (for example, if the patch is overly specific). And ensuring that the patch does not harm the program's normal functionality is even more difficult, as real world test suites are usually not sufficient to detect all regressions (e.g., if the vulnerability is in a component that is not actually exercised by the test suite, the repair system could "fix" the vulnerability by just removing the affected component—but this is unlikely to satisfactory to the developers). In the program repair community, these problems are known as *overfitting* (not to be confused with the conceptually similar use of overfitting in machine learning) and *weak proxies*, respectively, and are considered open problems [10]. In our benchmark, we use the proof-of-concept input found by the fuzzer as our vulnerability repair oracle, and (for now) do not attempt to check for regressions in program functionality. In our evaluation (Section 5), we manually validate the plausibility of a random sample of LLM-generated patches and report the fraction that fail to actually fix the flaw without introducing unintended side effects.

2.3. Large Language Models

Large language models (LLM) are large, with billions of parameters which are trained using machine learning methods. This training uses vast quantities of text, with an objective goal of predicting the most likely *completion* given a prefix string. Input text is typically converted to a sequence of integers (*tokens*) using a tokenizer using byte pair encoding (BPE) [11]; the number of distinct tokens available is called its *vocabulary* (generally 50K-100K for current models). The sequence of tokens is then fed to the model, which returns a probability distribution over the vocabulary representing the model's estimate of the most likely next token. To generate longer sequences, one can sample a token from this distribution, append it to the input, and feed this new input to the model to obtain predictions for the next token, and so on (up to the maximum number of tokens supported by the model—its *context length*). LLMs have been successful in a wide variety of natural language processing and program synthesis tasks over the past several years, and are widely used in commercial services like OpenAI ChatGPT [12], Anthropic Claude [13], and GitHub Copilot [14].

Because pure completion models are not very user friendly (e.g., it is difficult to give such models explicit instructions), models are usually fine-tuned by training on additional examples that demonstrate instruction following behavior (*instruction tuning* [15]) or updated using reinforcement learning based on user feedback (*reinforcement learning through human feedback*, or RLHF [16]). Fine-tuning can also be used to allow the models to edit text or code (e.g., by training on diff-style data) or act as interactive chat assistants (by training on dialogue data). In the evaluations we conduct in Section 5, we make use of both edit models (OpenAI's `code-davinci-edit-001`) and chat models (OpenAI's GPT-3.5 and GPT-4, as well as the open-source WizardCoder model).

3. Design and Implementation

We designed ARVO with the overall goal of producing a reliable, easy-to-use vulnerability repair dataset. In detail, we aim to achieve:

Quantity The dataset should contain a large number of vulnerabilities (thousands).

Quality Each vulnerability in the dataset should be validated to ensure it is actually a bug with security impact.

Diversity The vulnerabilities should be distributed across a large number of different projects, to ensure that evaluated repair systems are *general*.

Reproducible Each vulnerability should be reproducible; i.e., it should come with a triggering input that demonstrates the flaw, and a precise, developer-written patch that fixes the vulnerability.

Auto-Updating The dataset should automatically incorporate new vulnerabilities as they are found, to avoid issues with overfitting or memorizing answers to the benchmark.

Easy to Use The dataset should be easy for researchers and practitioners to use, without requiring them to have extensive security background or know how to build the projects in the dataset.

In the remainder of this section, we will describe how ARVO's design attempts to satisfy these requirements; in Section 4 we characterize the resulting dataset and demonstrate that it achieves these goals.

3.1. Overview

ARVO has a general architecture which is designed to ingest source metadata from 'bug'/project databases and augment this information with relevant source code, build steps, and binaries. Given the target purpose (automated program repair of C/C++ projects), the ARVO dataset also needs to include environments for re-compiling the source code so that modifications to the source code can be straightforwardly tested. To enable automated scripting, ARVO is also combined with an application programming interface (API). ARVO is thus separated into three major components, as shown in Figure 1: 1) the reproducer; 2) the vulnerability locator; and 3) the benchmark.

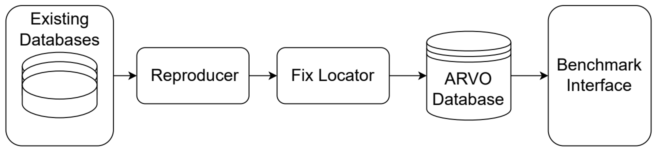

Figure 1: Overview of ARVO

The reproducer takes the provided metadata from the bug/project database(s) and compiles the project binary for the specified (vulnerable) version. Using the available version control information, we test that a given vulnerability was actually fixed by the canonical developer-provided patch by compiling the project both before and after the nominal fix, and then using the triggering (crashing) input to verify. Projects that do not pass this test are discarded.

The final step in the dataset construction is to locate the exact fix of each vulnerability. Because we have reproducible project build environments, we can iterate over the commit history for each project using binary search to identify the exact changes that fix the vulnerability. These change locations can be used both for repair localization as well as serving as a ground-truth reference for the fixes generated by evaluated repair strategies.

3.2. Source Data

In order to obtain a large quantity of vulnerabilities and allow the dataset to grow over time, ARVO is designed to draw project and bug metadata from upstream sources. This upstream data source is expected to provide some essential information, including version information, crash data, and details about the building environment.

- Version Information: This needs to include specific version identifiers (e.g., git commit hashes), enabling us to identify both the version containing the vulnerability and the version where it was fixed.
- Crash Information: This includes the inputs required to initiate crashes, logs from the crash state as captured by the sanitizer, and the necessary commands to execute the target software using the Proof of Concept (PoC) as input.

- Build environment: This refers to a virtualized, interactive environment capable of supporting the compilation and execution of fuzz targets.

These components are vital for accurately reproducing vulnerabilities and their fixes, as they allow for the compilation of fuzz targets, triggering of specific vulnerabilities, and verification of the applied fixes.

In our implementation, we selected OSS-Fuzz, the largest existing open-source vulnerability database satisfying our criteria, as our data source. This choice was made because OSS-Fuzz's public issue reports provide a comprehensive set of metadata. Each issue also features labels that can be used to select only bugs with security impact (`Type=Bug-Security`), that are reproducible (`label:Reproducible`), and for which a fix is available (`status:Verified`). Combining these query elements, we obtain over 10,000 issues in over 300 projects, which serve as the starting point for our dataset.

3.3. Reproducer

Bug and patch reproduction is the process of recompiling projects such that we can re-trigger crashes and inspect fix validity. Achieving universal reproduction, however, presents significant challenges. This task becomes increasingly difficult with older vulnerabilities, particularly when dependencies or toolchains have been lost over time. And although OSS-Fuzz does provide an official reproducer secript in its repository, it has not been updated for two years and has low rate of success at reproducing issues (13%, in a randomly selected sample of 100 vulnerabilities). However, it serves as a useful starting point for our own reproduction efforts. Using the techniques described in this section, ARVO's updated Reproducer component is able to increase the success rate to 37% in that same sample, and ultimately reproduce 3,479 vulnerabilities out of 9,336 vulnerabilities (37.3%).

Version Control. A critical challenge in recompiling older projects lies in ensuring the accuracy of version control. The precise timestamp is essential for utilizing the correct building environment and version-specific components.

The building environment provides the necessary toolchain, which plays a pivotal role in reproducing the vulnerabilities. For example, newly introduced warnings or more strict checks in more recent compilers may cause older code to fail to compile. Also, a build script written 4 years ago might not be compatible with the latest version of the compiler. Thus, accessing specific versions of these toolchains is crucial. Luckily, the OSS-Fuzz infrastructure maintains archived Docker base images that capture the different toolchains used throughout the project's history in `gCloud`. We rely on the timestamp provided in the issue metadata to locate the correct toolchain. When doing so, we found that taking into account time zone information was critical, as the toolchain was sometimes updated multiple times in the span of a few hours; without time zone adjustments many projects failed to build, and we were able to achieve an 11.11% increase in the reproducer's success rate by fixing this issue.

Additionally, dependency version control is also very important. To retrieve the build script, we had to access OSS-Fuzz's GitHub repository and roll it back to the version in use at the time the vulnerability was discovered. Initially, we matched old versions using the timestamp of the vulnerability report, but this led to several compilation errors due to missing files. This mismatch between OSS-Fuzz's build script and the source code suggested a deeper issue. On further examination, we found that the project frequently modified its build script on the reported day, necessitating a more precise timestamp than the one provided in the report. However, we were able to retrieve more accurate timestamps from the OSS-Fuzz `gCloud` archive, allowing us to gain an additional 25% improvement in reproducer success.

Finally, the standard tools provided by OSS-Fuzz primarily focus on the main project component, often defaulting to the latest versions for other dependencies. This approach, while convenient, fails to account for compatibility issues between the main component and its dependencies. For instance, the project `imagemagick` relies on 15 separate components, each with frequently changing APIs and usage patterns. Consequently, operating an older version of `imagemagick` with the latest versions of these 15 dependencies is challenging. This mismatch not only causes compatibility issues but also leads to inconsistencies between the dependencies and the compiling environments, as previously discussed in the version control section. We again account for this by identifying the precise timestamp for each vulnerability and ensuring that the versions of each dependency are rolled back to the version in use at that time. This improved the reproducer success rate.

These challenges, involving often-changing build toolchains, project build scripts, and dependencies, highlight the critical role of precise timestamping for the reproduction of vulnerabilities. They also point out areas where vulnerability databases like OSS-Fuzz could be improved, e.g. by capturing more precise version information not only about the main project but also its associated toolchain and dependencies.

Broken Resource Fixing. In our efforts, we encountered numerous dependencies where components were no longer accessible, particularly for projects from the 2017-2019 period. During this time, many projects migrated their repositories from Subversion to git, rendering previous methods for sourcing these components obsolete. Additionally, certain build scripts rely on resources downloaded from the Internet, which may become unavailable over time.

Some missing resources could be crucial for reproduction since any error during compiling is fatal. For instance, specific online tool versions may be necessary to compile key components, and the developer hardcoded the URL in the `Dockerfile`. After years, the URL expired so we have no way to get such a tool while building the docker image from the given `Dockerfile`. Alternatively, some resources might be relevant only for fuzzing rather than

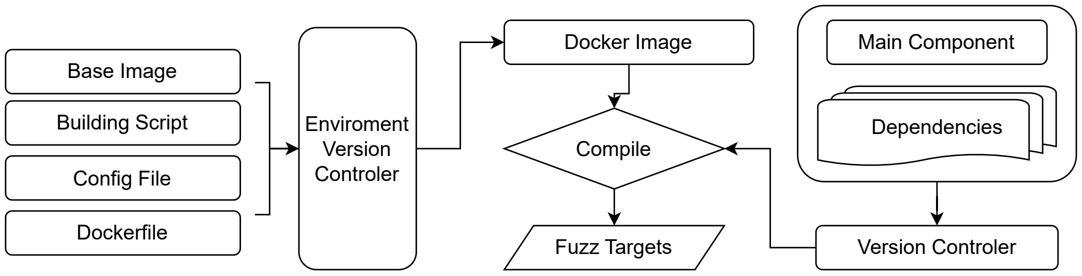

Figure 2: ARVO Reproducer Structure

actual compilation. During our implementation process, we categorized these issues on a project-by-project basis. We then analyzed each expired resource and divided them into two categories: core and non-core resources.

For core resources (e.g., dependencies), which are essential for compiling the fuzz target, we endeavored to manually locate the missing resources, prioritizing projects based on the number of vulnerabilities associated with the project. We manually inspected the top 20 such projects, found their missing resources (e.g., by replacing Subversion repository URLs with the corresponding git repository); we fixed 32 missing dependencies in total, allowing us to attempt reproduction of vulnerabilities in the projects that depended on them. Meanwhile, for non-core resources, which don't impact the compilation process for the fuzz target itself (but which cause other parts of the build to fail, such as documentation generation), we simply identified and removed the related code from project's build system to prevent these resources from disrupting the compilation process.

3.4. Fix Locator

Patch information is used to provide the localization oracle for our benchmark, as the ability to pinpoint the precise fix for a given vulnerability is important. However, this data is not directly provided by OSS-Fuzz. To solve this issue, we include in ARVO a dedicated Fix Locator component; our Fix Locator is able to identify precise fixes for vulnerabilities in 83.6% of the reproducible cases in our database. We believe that these developer-written patches for vulnerabilities may also be of interest in their own right for future research in vulnerability repair.

Challenge. When OSS-Fuzz finds a vulnerability, it monitors the project repository and will periodically check whether the vulnerability is fixed. However, this check is only performed once a day, which means that the OSS-Fuzz report only identifies a range of possible revisions where the fix may have occurred, rather than identifying the exact patch. Thus, although ARVO starts with the provided OSS-Fuzz data, it then conducts a search for the exact revision where the vulnerability is first patched.

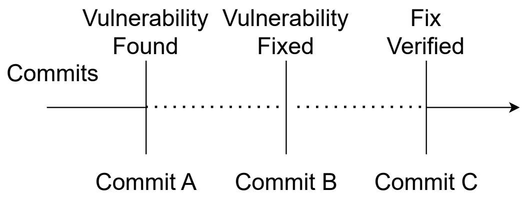

Figure 3: The Vulnerability lifecycle in OSS-Fuzz

Figure 3 illustrates the process projects pass through when involved in fixing a bug discovered by OSS-Fuzz. The process begins with Commit A, where OSS-Fuzz detects a crash and informs the project developers about the vulnerability. The developers then identify the root cause and commit a fix for the issue (Commit B), which can be time-consuming (vulnerabilities in our dataset averaged 68.89 days between the initial report and the fix). During this period, numerous unrelated commits might occur between Commit A and Commit B; fixes are also often made in a separate branch and later merged into the main branch. These factors complicate the search for the exact fix.

After the vulnerability is addressed by the developers in Commit B, OSS-Fuzz will (after some delay, during which other commits may be made to the project) reviews and officially confirms the patch of the vulnerability in its report. This verification occurs at Commit C. OSS-Fuzz reports only Commits A and C, so we need to do additional work to search for the Commit B, which actually fixed the flaw. This sequence highlights the complexities involved in tracking and verifying fixes for vulnerabilities, especially considering the potential for multiple non-related commits and the practice of using separate branches for patches.

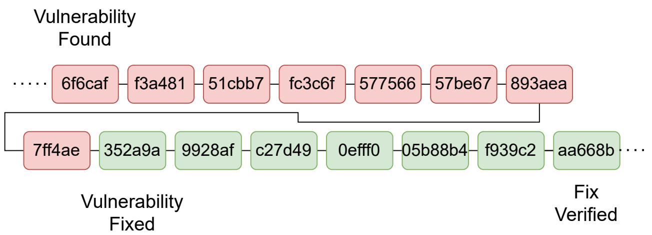

Figure 4: Open Issue 44851 on OSS-Fuzz

Figure 4 shows a concrete example of the lifecycle of an OSS-Fuzz vulnerability: issue 44851 is a vulnera-

```
--- a/MagickCore/draw.c
+++ b/MagickCore/draw.c
@@ ... @@ static double GetFillAlpha(...
    i=(ssize_t) MagickMax...
-   for ( ; I < p->number_points; i++)
+   for ( ; i < p->number_points-1; i++)
    if ((double) y <= p->points[i].y)
        ...
```

Figure 5: The fix identified by ARVO for issue 44851.

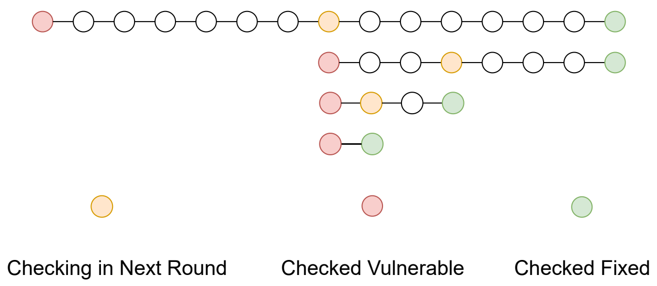

Figure 6: Locator

bility (specifically, an off-by-one heap buffer overflow) in `Imagemagick`. According to the OSS-Fuzz report, the bug was identified in the `imagemagick` GitHub repository at commit hash `6f6caf`, with the developer reportedly fixing it the following day at commit `aa668b`. However, a closer examination reveals that commit `6f6caf` only involved a minor string modification in the file `ChangeLog`—it is not the actual fix for the vulnerability. Between the time the vulnerability is found and when OSS-Fuzz verifies the fix, there are a total of 14 different commits, affecting 83 files, making it difficult to determine exactly where the real fix occurred by hand.

Solution. To locate the precise fix, we conduct a binary search over all commits (across all branches) in the range reported by OSS-Fuzz. At each step in the binary search, we use the reproducer test case to check if the vulnerability is fixed; if it is, then we proceed to search earlier commits, and if not, we continue the search with later commits. When the binary search terminates we have identified the earliest commit where the vulnerability is no longer present, which should be the actual patch. We note that this search is possible because each vulnerability comes with a triggering input, and we can build and run the target project at each revision in the range automatically. If this does *not* hold (i.e., if there are some revisions in the range that can't be built, either because of toolchain issues or because the developers committed non-compiling code) then we will not be guaranteed to find the precise fix. We therefore treat build errors during the binary search as fatal and report that we cannot locate a fix.

As a concrete example of our search procedure, consider the case illustrated in Figure 4. ARVO locates the fix at commit `352a9a` after four steps. We manually verified that this commit (shown in Figure 5) is indeed the patch that fixes the heap buffer overflow in `Imagemagick`.

3.5. Benchmark API

A goal of our benchmark is that it should be easy to use, even for researchers who do not have a security background; we hope that this will allow researchers in other fields, particularly machine learning researchers, to use it as an evaluation target.

To achieve this, we created pre-built Docker images (exported as `tar` archives) for each vulnerability that contains the compiled project and all its dependencies at its vulnerable and fixed versions, along with a `bash` script that runs the proof-of-concept input inside the container on the target program. To reproduce a vulnerability or validate its fix, end users can download the Docker image, load it (via `docker load`), and run the provided script. The images in our dataset currently total 30TiB (but can be downloaded individually to evaluate subsets of the dataset).

We also designed the benchmark API to handle the details of making the actual source changes, rebuilding the project with the modified code, and testing for the presence of the vulnerability.

For a given vulnerability local identifier, ARVO provides a report about the vulnerability containing its metadata, including the project information, proof-of-concept input, the developer patch identified by ARVO, and source code of the vulnerable function[1]. Benchmark users can then use that report to generate their proposed fix, and have it automatically tested by running `tryFix ID fixed_Code`; this outputs a log with the overall result (true if the updated code fixes the vulnerability, false otherwise) along with any output from the build process and output from the program while processing the proof-of-concept input.

To support more advanced uses of the dataset (e.g., rebuilding the project with other instrumentation), we make ARVO itself open source so that researchers can rebuild the Docker images from scratch with their desired changes.

4. Dataset

In this section, we present the details of the resulting ARVO dataset constructed using the methods described in Section 3.

4.1. Dataset Characteristics

4.1.1. Dataset Size and Growth. At the time of this writing, out of 9,336 vulnerabilities initially obtained from OSS-Fuzz, we were able to reproduce 3,479 vulnerabilities across 231 projects, and precisely locate the associated fix for 2,908 and their associated fixes across 211 projects. Figure 7 shows the proportion of OSS-Fuzz vulnerabilities we are able to reproduce and locate fixes for over time; we

1. Only the vulnerable function is provided from the API; however, if the repair system needs additional information, such as other project source files, they can be obtained from the Docker image. Repair systems that make use of dynamic analysis can also run the program in the container and extract runtime information.

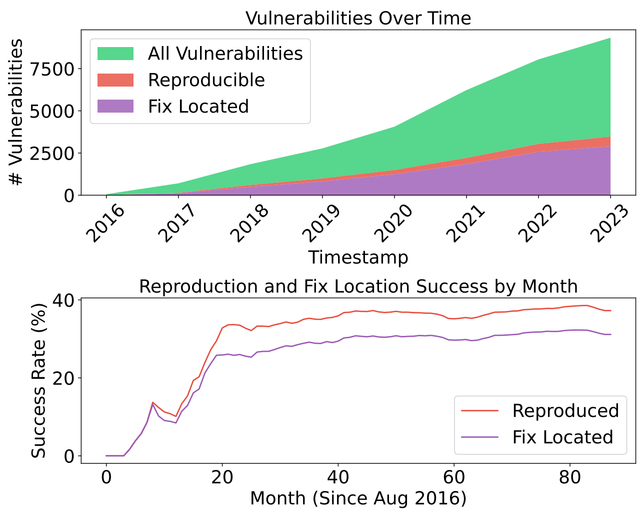

Figure 7: Database Growth

can see that older vulnerabilities have significantly lower success rates. This is both because older vulnerabilities are more likely to depend on resources that are now missing, and because the older issues in the OSS-Fuzz bug tracker use an older report format that lacks some important metadata such as dependency versions. On more recent vulnerabilities, our success rate has held steady at around 35% for reproducing the vulnerability and around 30% for locating the corresponding fix.

4.1.2. Project and Language Distribution. Table 1 shows the distribution of vulnerabilities among projects. This distribution is relatively even; the project with the most vulnerabilities represents 7.6% of the dataset, and the top 10 projects collectively account for only 31%. This means that repair systems cannot simply focus on a handful of projects with a disproportionate number of vulnerabilities, but must instead handle a wide range of different projects.

In terms of languages, Table 2 shows the vast majority (95.7%) of the projects in the dataset use C (50) or C++ (174), but there are also a handful of Rust (3), Python (2), and Swift (2) projects. C and C++ dominate because ARVO currently focuses on low-level memory safety issues, but we hope to improve extend it to include other languages supported by OSS-Fuzz in future work.

4.1.3. Patch Statistics. Here we provide an analysis of the developer-provided patches in our dataset. Out of the 2,908 vulnerabilities in ARVO for which we were able to identify the precise commit fixing the vulnerability, we filter out duplicates (which can occur when a single patch fixes multiple vulnerabilities) and remove merge commits (since these contain many changes unrelated to the vulnerability). The filtered set contains 2,150 patches.

We find that the average patch fixing a vulnerability affects 2.98 files (mean: 2.98, median: 1, std: 16.13); 1,386 patches (64.4%) affect just a single file. This accords with prior research finding that security-related fixes are typically small and self-contained [17]. Looking at the number of lines added and removed by each patch, we find a median of 6 lines added and 2 lines removed; the means of both are significantly larger (161.1 added and 120.8 removed) due to a small number of outliers. 90% of the patches in our dataset have fewer than 60 lines added or removed.

4.1.4. Simple Flaws Subset. Because current LLM-based repair systems have limited context windows, we also identify a subset of the vulnerabilities in our dataset whose root cause and fixes are small and self-contained. This subset is chosen by analyzing the developer patches selecting vulnerabilities that affect just a single function in the target program. These resulting 1,377 “simple” cases are used in our evaluation (Section 5) to provide baseline evaluations of current LLMs.

Project	# Vulnerabilities	% of Dataset
imagemagick	265	9.11
harfbuzz	139	4.78
ghostpdl	134	4.61
ndpi	128	4.4
binutils-gdb	97	3.34
opensc	73	2.51
wireshark	71	2.44
c-blosc2	70	2.41
libxml2	64	2.2
openthread	61	2.1
Other Projects	1806	62.1

TABLE 1: Project Distribution in ARVO

Language	# Vulnerabilities	% of Dataset
C++	174	75.3
C	50	21.6
Rust	3	1.3
Python	2	0.9
Swift	2	0.9

TABLE 2: Language Distribution in ARVO

4.2. Comparison with Other Datasets

ARVO is not the first dataset that aims to be useful for evaluating vulnerability repair, but we believe it is the largest that satisfies the goals we set out in Section 3. In Table 3, we compare ARVO qualitatively and quantitatively to previously published vulnerability datasets.

From the table, we can see that ARVO is the *only* dataset to provide a proof-of-concept (PoC) input the ability to automatically build the vulnerable software in order to test it. The former allows repairs to be quickly validated by running the program on the PoC input, while the latter simplifies the task of getting the project and its dependencies running. Meanwhile, the ability to automatically update the dataset as its upstream data source reports new vulnerabilities (a feature shared with CVEFixes [20]) means that it does not run the risk of having repair systems overfit to the benchmark or (in the case of machine learning-based systems)

TABLE 3: Comparison of ARVO with related datasets.

Dataset	# Vulns	# Projects	PoC	Auto-Update	Auto-Build	CVE/CWE	Real-World
ARVO	2,908	231	✓	✓	✓	✗	✓
Big-Vul [18]	3,754	348	✗	✗	✗	✓	✓
ExtractFix [19]	30	7	✗	✗	✗	✓	✓
CVEFixes [20]	5,495	1,754	✗	✓	✗	✓	✓
CGC	276	249	✓	✗	✓	✗	✗

having all of its vulnerabilities and fixes appear in the ML model's training data. Finally, while the DARPA Cyber Grand Challenge (CGC) dataset has excellent automation and provides PoC inputs, it has a relatively small number of programs, each with only 1-2 vulnerabilities; moreover, the programs were created specifically for the CGC event and use DARPA's custom DECREE operating system, limiting the real-world applicability of the dataset.

The only area where ARVO falls short is in associating each vulnerability with a CVE (Common Vulnerabilities and Exposures) or a CWE (Common Weakness Enumeration), identifying the specific vulnerability or class of weakness, respectively. This is because the vast majority of vulnerabilities found by OSS-Fuzz are never assigned a CVE: the fuzzer finds the flaw, automatically reports it, and the developers fix it within the span of a few days. In future work, we hope to build automated tools to help automatically categorize each vulnerability in the dataset and assign it a CWE class.

5. Evaluation

In this section, we conduct comprehensive baseline evaluations of popular Large Language Models (LLMs) including `GPT-4`, `GPT-3.5-turbo`, `GPT-3.5-turbo-16k`, `Code-davinci-edit-001`, `Wizard-34B-V1.0`, `Wizard-15B-V1.0`, and `StarCoder`, with the ultimate objective of establishing ARVO as a reliable benchmark dataset. We also analyzed these results to identify strengths and limitations of current LLMs in vulnerability repair. Finally, we manually analyze several case studies among the LLM-generated fixes to provide a qualitative assessment and identify directions for future research in LLM-based vulnerability repair.

5.1. Overview

We conduct three main evaluations: 1) We evaluate the breadth of our dataset by measuring the performance of `gpt-3.5-turbo-16k` across the full "simple" subset of ARVO described in Section 4.1.4; 2) We provide a comparison *between* LLMs on a sample of 100 vulnerabilities; and 3) We test whether LLM performance on the dataset is due to training data contamination by sampling 100 vulnerabilities before the models' training cutoff and 100 after, and then comparing the two. For each experimnent the primary quantity we measure is success at fixing the vulnerability; we do not attempt to quantify whether the fix degrades unrelated program functionality. For all models, we use a temperature of zero to obtain deterministic results.

5.2. Evaluation Setting

While conducting our experiments with various AI models, the construction of prompts played a crucial role in standardizing the input and ensuring the consistency of the test cases. Each prompt was formulated with a three-part structure: an instruction, the vulnerable code, and a patch description. The instruction is a way to set the tone on what the model should do/produce, such as "Rewrite the code." We then proceed to append the vulnerable function and the patch description. In order to mimic the situation in which researchers performed the comprehensive analysis of the crash, we assume the researchers already had basic information about the bug before asking AI models to fix it. In the evaluation, we produced a patch description of each vulnerable case by inferencing the GPT-4 model to explain the potential fix in words given in the git diff file (see the Appendix, Listing 4). To avoid providing too much information that could be hard to get, we only provide a short description of AI models to provide a baseline result, and all these descriptions can be found in our evaluation log data. Meanwhile, considering the inconsistency of the patch description returned from GPT, we also performed the same evaluation for "Lite-Prompt," which means we only used the crash type from the sanitizer. The result of Lite Prompt could be a baseline for people using AI models to fix vulnerabilities.

Acknowledging the fundamental differences in the AI models' operational paradigms is crucial, particularly between GPT (Generative Pre-trained Transformer) and Codex models. GPT models, such as GPT-3.5 and GPT-4, are primarily designed for completion tasks. They excel in generating text that logically and coherently follows from a given prompt. On the other hand, Codex models are tailored for editing tasks. They are adept at understanding and manipulating existing code to achieve a specified objective, such as bug fixing or optimization. Due to the inner differences between them, we have to design different prompts for them. However, we followed the least-difference rule to decrease the influence of the different interaction methods and ensure the formatting of the prompts and the information they got was similar across different models.

The full prompt and the role setting for each model can be found in Appendix A along with the patch-description setting.

5.3. Simple Fix Case Evaluation

Before delving into the comprehensive benchmarks of various AI models, it's important to highlight a specific test conducted with gpt-3.5-turbo-16k, focusing on a subset of what we classified as ”easy” cases within the ARVO dataset. For this experiment, we selected 1,377 cases characterized by their relatively straightforward nature, where each vulnerability’s fix was confined within the boundaries of a single function.

This test was designed to evaluate the capability of gpt-3.5-turbo-16k in handling simpler, more contained repair scenarios, offering a baseline for its proficiency in software vulnerability patching.

Due to the time limit and resource cost issues, we’ll not test all the models on the selected dataset. After we evaluate these 1,377 cases, we’ll set a selection range for further benchmarks.

Following the setting in the previous section, we finished the 1,377-case evaluation and got the result in Figure 8. The overall fixing success rate of gpt-3.5-turbo-16k is 21.2% (292/1,377), but we can see the success rate improvement as time goes on. As Figure 8 shows, we sorted 1,377 cases by time and then split them into three equal parts, and each part included 459 cases. To demonstrate the trend, we mark them in three colors and see that the fixed success tally number increases over time.

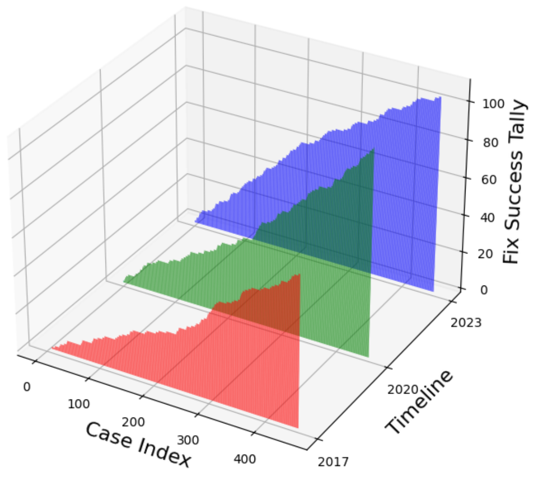

Figure 8: 1,377 Simple Vulnerabilities Fixing Result

Considering gpt-3.5-turbo-16k ended its training in September 2021, this trend is unexpected. gpt-3.5-turbo-16k should work better on bugs before September 2021, but our result shows gpt-3.5-turbo-16k worked better in more recent data.

Based on our observation, we assume even though our testing bugs may appear in gpt-3.5-turbo-16k’s train data, they performed very little influence on the model since we found there is not a clear cut on the graph that shows the train influence on gpt-3.5-turbo-16k. The abnormal trend is likely due to recent materials being more frequently represented in training data. Also, this interesting trend implies that the model’s ability to patch vulnerabilities is more likely attributed to its understanding of coding patterns and principles rather than directly recalling previously encountered cases.

Even training seems to have very little influence on fixing old bugs. To avoid this influence, we only chose the bugs after September 2021, which was later than the training stop date for GPT-4, GPT-3.5-turbo, GPT-3.5-turbo-16k, and Code-davinci-edit-001.

Based on this rule, we randomly selected 100 cases and verified GPT-3.5-turbo-16k’s success rate on them(23%) was very close to the overall success rate for post-train cases(24.3%). This selection of the test dataset avoided the influence of training data and showed good representativeness.

5.4. Multiple Evaluation over the same Dataset

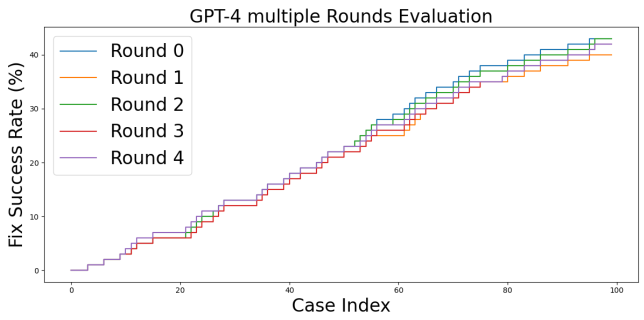

Figure 9: GPT-4 Performance for Multiple Rounds

In the previous subsection, we established a representative test set to benchmark but still need to prove the soundness of our benchmark result. In order to show our benchmark’s result can stably show the performance of the AI models, we used GPT-4 to fix 100 selected vulnerabilities multiple times. We compared the results to see if the AI model could understand the code or if it was performing random patches. The result in Figure 9 shows GPT-4 has a pretty stable performance for each round, represented by the growth of the lines. These lines shared almost the same growth, which means they succeeded in similar vulnerabilities. In detail, 37 cases were successfully fixed in all five rounds, and each round fixed 43, 40, 43, 42, and 42 cases, respectively. We computed its coefficient of variation by $CV = \frac{\sigma}{\mu} = 0.026$, which also shows the result is quite stable.

Based on this result, we proved GPT-4’s fixes are based on its understanding of the code. The result of fixing is also reproducible. Moreover, using the ARVO database as the benchmark tool can evaluate others’ work honestly.

5.5. AI Model Benchmarks

The introduction of the ARVO dataset marks a significant milestone in the realm of AI model benchmarking, particularly in the domain of automatic vulnerability repair. As a comprehensive and diverse repository of software vulnerabilities, ARVO provides an unparalleled platform for rigorously testing and evaluating AI models. By leveraging ARVO, researchers and developers can gain deeper insights into how different AI models perform in diverse and complex real-world scenarios, paving the way for advancements in automated vulnerability repair techniques.

We present the results of our comprehensive benchmarks using ARVO, evaluating the performance of various AI models in software vulnerability repair. The AI models we assessed include `GPT-4`, `GPT-3.5-turbo`, `GPT-3.5-turbo-16k`, `Code-davinci-edit-001`, `Wizard-34B-V1.0`, `Wizard-15B-V1.0`, and `StarCoder`. To gauge the effectiveness of these models, we adopted a dual-prompt approach for each case across two separate sessions. In the first session, the models were provided with a more detailed vulnerability description generated by `GPT-4`. In contrast, they were only given crash-type information grabbed from sanitizer in the second. This approach allowed us to assess the models' capability to interpret and respond to varying levels of detail in the prompts.

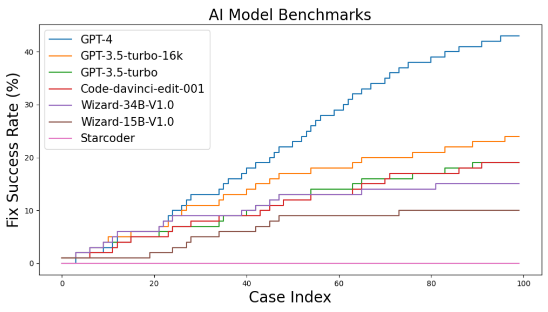

Figure 10: AI Model Benchmarks

First Session. In the first session, as the result in Figure 10 shows, with enough information in the vulnerability description, most models can fix plenty of vulnerabilities. The `GPT-4` model showcased superior performance; `GPT-4` achieved a remarkable success rate by patching 43 out of the 100 cases. For the second rank, `gpt-3.5-turbo-16k` successfully patched 24 cases. Following, `GPT-3.5-Turbo` and `Code-davinci-edit-001`(Codex) both got 19 cases successfully fixed while `Wizard-34B-V1.0` and `Wizard-15B-V1.0` fixed 15 and 10 vulnerabilities, respectively. Unfortunately, `Starcoder` failed to fix any vulnerability in our dataset.

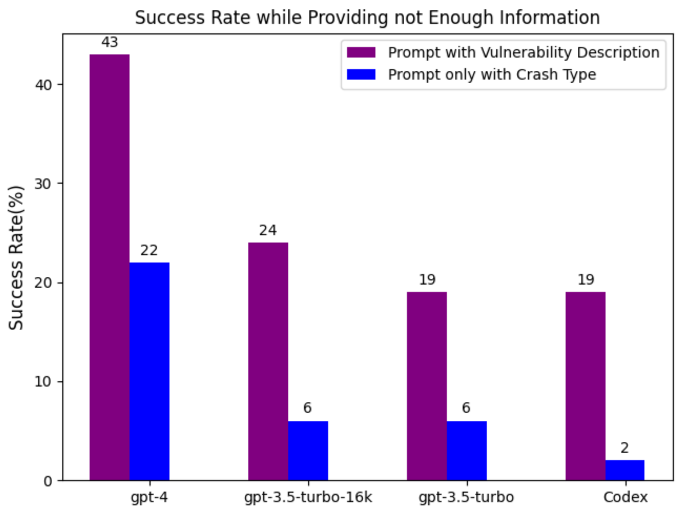

Figure 11: Simple Prompt Results in Worse Success Rate for Fixing

Second Session. The second session presented a more challenging scenario, where models were only given crash information. `Codex` patched two vulnerabilities here, while `Wizard-15B` and `Wizard-34B` repaired 4 cases each. `GPT-3.5-turbo`'s performance dropped to patching 6 cases. `GPT-4`, although impacted by the reduced information, still performed notably; `GPT-4` successfully repaired 22 vulnerabilities. The more advanced models, particularly `GPT-4`, demonstrated a significant proficiency in dealing with complex repair tasks, especially when provided with detailed instructions.

Result. The experiments conducted in both sessions demonstrate the variability in model performance, which depends on the nature and detail of the prompts and the models themselves.

In the first session, with detailed vulnerability descriptions, the AI models showed a relatively high success rate in patching vulnerabilities. `GPT-4` emerged as the top performer, successfully addressing 43 out of 100 cases. This was a significant achievement compared to other models. `gpt-3.5-turbo-16k` was able to fix 24 cases. `GPT-3.5-turbo` and `Codex` both repaired 19 cases. Though lower, the performance of `WizardCoder-34B-V1.0` and `WizardCoder-15B-V1.0` models still demonstrated their capability by fixing 15 and 10 cases, respectively. The inability of `Starcoder` to fix any vulnerabilities in this session highlights the varying levels of sophistication and applicability among different models. In the more challenging second session, which relied on minimal crash information, there was a notable drop in the success rates across all models. Nevertheless, `GPT-4` still managed to repair 22 vulnerabilities, showcasing its resilience in less informative scenarios. `gpt-3.5-turbo-16k` and `gpt-3.5-turbo` achieved the shared record of 6 successful patches. `WizardCoder` models both achieved four successful fixes, and `Codex` was able to only fix 2 cases. This session further illustrates the impact of prompt detail on their vulnerability repair capabilities.

Comparing `gpt-3.5-turbo` and `gpt-3.5-turbo-16k`, we observed that `gpt-3.5-turbo-16k` outperformed `gpt-3.5-turbo`. This difference can be attributed to `gpt-3.5-turbo-16k`'s ability to analyze longer cases, highlighting the importance of input length in handling complex vulnerabilities. The number of maximum output tokens also plays a role. `WizardCoder` models, limited by default maximum output token length of 2048, showed lower performance. This limitation underscores a crucial aspect of vulnerability repair: the need for comprehensive and detailed input to identify and address issues that may span beyond a single function.

The second session, with minimal crash information, further highlights the impact of prompt detail on model performance. The shared achievement of `gpt-3.5-turbo-16k` and `gpt-3.5-turbo`, and the overall drop in success rate compared to the first session in this more challenging environment reinforces the significance of detailed information for an effective repair. The session also suggests that models like `Codex`, though advanced, may require more explicit information to function optimally in vulnerability repair.

These findings present a baseline for future work, where the location and specificity of the vulnerability information can be optimized for better results. For instance, in cases of out-of-bounds (OOB) errors, pinpointing the specific variable involved could significantly enhance the repair process. This insight from the 'Not Lite Prompt' session emphasizes the importance of detailed, overall descriptions of bugs for AI models to function effectively regarding vulnerability repair.

Furthermore, the results from these benchmarks prove that the ARVO dataset is a powerful tool for future research in this area. ARVO's diversity and scale have enabled a clear demonstration of the expected differences in performance between various models, which establishes ARVO as a crucial resource for further exploration and development in AI-driven automatic vulnerability repair. The findings from these benchmarks not only contribute to our understanding of the current state of AI in software security but also pave the way for future advancements, ensuring more effective and reliable AI solutions in the ever-evolving landscape of software vulnerability management. However, we need to ensure the large language models weren't simply lucky to fix the vulnerabilities. It is important to delve deep into analyzing sample fixes produced by the models. This exploration aims to verify the quality and reliability of the repairs made by the model, thus providing a more comprehensive understanding of its ability and potential areas of enhancement.

5.6. Case Studies

In the potential areas for vulnerability fixing benchmark, it is crucial to test whether the fix generated by the AI model is "plausible." This analysis compares the fix produced by `GPT-4` with the original fix, shedding light on the AI model's understanding and approach to vulnerability repair in C/C++ code. We delve into a detailed case study from the ARVO dataset `localId=44695`.

Prompt Format and System Role. As mentioned in section 5.2, we use similar prompts for different AI models to ensure they'll get similar information to fix the vulnerability. The prompt format is crucial in guiding the AI model's response. We set the system role of the model with the following initial instruction 2. Following this, we present the instruction prompt, which includes the vulnerable code and a patch description.

We used the vulnerability description generated by `GPT-4`, which is generated by instructing the model to provide information about 1) What type of vulnerability it is, 2) What situation it is triggered, and 3) What is the related variable used for 3. The description generated is used to provide sufficient information to the models.

As discussed earlier, providing sufficient information to the models heavily influences the quality of the fixes. For Out-of-Bounds vulnerabilities, by leveraging crash information and context from sanitizers, we are able to retrieve the context information at the point the program crashes, which is helpful for getting more precise crash types. We can extract additional details from backtraces during crashes, further equipping the AI models with comprehensive data needed for effective repair.

Comparative Analysis of the Fixes. In the case of `localId 44695`, the Git diff 4 file reveals `GPT-4`'s approach to the fix. `GPT-4` made two modifications in the function. The location of the first modification is the same as the real world. While other models like `gpt-3.5` models, `Codex`, and `WizardCoder` models either failed to identify the correct modification area or exceeded output token limits, `GPT-4`'s ability to pinpoint the vulnerable spot and propose a fix, albeit with additional changes, sets it apart from its counterparts.

However, `GPT-4`'s inclusion of an extra, unrelated change within the same function raises critical questions about its understanding of the code, despite being given a detailed patch location such as "`where the code does not correctly check the bounds when parsing a floating point number.`" This observation suggests that while GPT-4 can generate syntactically correct and seemingly relevant fixes, its comprehension of the underlying issue and the minimal necessary changes might not always align with the optimal solution.

Hybrid Analysis Method. We noticed that the third point of patch description is hard to get based on traditional analysis methods, such as root cause analysis, dynamic analysis, and static analysis. But our prompt generator(GPT-4) got that from the source code (patch). By manually verifying the statement made by the prompt generator, we found it corresponded to the correct usage of the variable. This inspiring finding hints at the potential of combining traditional crash analysis methods with AI models to deliver more insight analysis. The traditional analysis provides basic and more precise information, while the AI models extract deeper and less confident results about the context. With these two parts of information, we can better apply AI models to fix the vulnerabilities. This idea also matches the principle of guiding AI to solve the problem step by step.

Pipeline the Fixing. Based on the observation that different modules may be good at different scenarios, we propose constructing a pipeline to integrate multiple models effectively. This pipeline would connect different AI models, leveraging their strengths to create a more robust and comprehensive vulnerability repair system. In this setup, the output of the best-performing model for a given scenario (e.g., `GPT-4` in our case studies) would be reviewed by other models for validation. This cross-checking process could serve as a quality control mechanism, ensuring that the proposed fix is not only technically sound but also contextually appropriate.

This approach may significantly enhance the accuracy and reliability of automatic vulnerability repair. By combining the capabilities of different models, the pipeline can mitigate individual model limitations, such as overfitting or lack of understanding of complex code structures. For instance, a model that excels in identifying the correct location for a fix could be paired with another better at generating minimal and precise code changes. Furthermore, this pipeline could also incorporate feedback mechanisms, allowing models to learn from each other's successes and failures, thus continuously improving their repair strategies.

6. Limitation

Our methodology faces certain limitations. In reproducing vulnerabilities, we did not verify if the crash type and address matched the reported vulnerabilities, potentially affecting the accuracy of our reproductions. Additionally, our database includes duplicated bugs from OSS-Fuzz, which we did not filter out. This oversight means our database is smaller than reported, impacting the robustness of our evaluation results.

Furthermore, our reliance on binary search for identifying vulnerability fixes has limitations. Due to possible multiple related commits, this approach might not always accurately pinpoint the exact fix, particularly when fixes involve a series of modifications. Also, while a commit may lead us to a presumed fix, it might encompass extensive modifications, complicating the identification of the precise change responsible for the fix.

Another concern is our omission of functionality checks on the implemented fixes. Our observations suggest that most fixes while avoiding the trigger of the vulnerability, may not be functionally optimal or might simply remove the vulnerable code block without addressing the underlying issue.

7. Future Work

To enhance the effectiveness of our research, we aim to improve the success rate of locating and reproducing vulnerabilities. A higher success rate would not only expand our database but also accelerate its growth, which is crucial for advancing security and AI research. Addressing the version control challenges highlighted earlier is a key aspect of this effort. We plan to refine our methodologies, drawing inspiration from the development of ARVO, where addressing seemingly minor engineering problems yielded significant improvements.

In addition, we intend to integrate ARVO's features with OSS-Fuzz and ensure ARVO's continual updates. Given the advantages of archiving vulnerabilities at the time of their discovery, our focus will also include developing a pipeline for assimilating the latest published vulnerabilities and enhancing the independence of our compiling processes.

8. Related Work

8.1. Automated Program Repair: Methods

The field of automated program repair is large, with many high quality surveys presenting the state of the art [8], [21] including the living review by Monperrus [22]. Automated vulnerability repair is a subset off this area which focuses solely on repairs to code causing security violations. These are typically built on techniques and tools for detecting security bugs such as those listed by OWASP e.g. CodeQL, etc. [23], [24], [25], [26]. A repair here refers to a fix to a program or source code currently violating a specification (which can be security-focused) [19]. Approaches can be based on symbolic execution and formal properties [19] and matching code to known weak design patterns/signatures [27], [28].

A number of techniques based on machine learning for automated vulnerability repair have been suggested. These are typically based on training models from faulty examples such that they can generalize to faulty but unfamiliar code, e.g. learning vulnerable abstract syntax tree patterns [29], learning vulnerable API patterns in Java [30]. Neural machine translation (NMT) techniques train dedicated models (such as recurrent neural networks) over bug/-patch pairs [31], [32], [33], [34]. However, these approaches typically are limited to simple, single-line fixes or specific languages, and still require the curated datasets such as that presented in this work.

Meanwhile, using off-the-shelf transformer-based large language models (LLMs) for automated vulnerability repair was explored by Pearce et al. [1], but although they studied a range of LLMs (including a custom decoder-only model) their study used only a few (12) real world examples from the ExtractFix dataset [19]. Chen et al. [35] trained a small encoder-decoder model 'VRepair' for repairing C functions, and Fu et al. [36] trained a larger encoder-decoder T5-based LLM which had more capabilities in this space. Other types of transformers may also be used for vulnerability repair, and Fu et al. have explored using vision transformers in this manner [37], finding that a 'vulnerability mask' can be trained into self-attention mechanisms to boost identification of vulnerable program areas. For more information on LLMs targeting program repair, Zhang et al. has conducted a survey [38].

8.2. Automated Program Repair: Datasets

Prior work has introduced a number of datasets that can be used to evaluate vulnerability repair [19], [20], [35], [36]. As we discuss in detail in Section 4.2, most of these datasets either have a relatively small number of flaws, do not provide a PoC input that can be used to automatically test repairs, or are manually constructed and difficult to update with new vulnerabilities. Finally, one prior work has attempted to make the task of running repair evaluations easier by providing a distributed REST interface to benchmark datasets [39], similar in spirit to the ease of use ARVO provides through its benchmark API.

9. Conclusion

In this paper, we present ARVO - an Atlas of Reproducible Vulnerabilities in Open source software, established as a scalable and accessible benchmarking tool for binary security and LLM-based security research. This paper outlines our approach for transforming a document-centric vulnerability database into an interactive benchmarking tool. Additionally, we identified and solved numerous challenges associated with building and reproducing historical vulnerabilities, despite the "bit rot" of their associated dependencies, resources, and toolchains. Given we expect this paper will be useful for development of LLMs for security-focused tasks such as automated vulnerability repair, we provide baseline experiments and results for major LLMs including ChatGPT and Codex. Our dataset of reproducable vulnerabilities is open-source, contains nearly 3,000 vulnerabilities across more than 200 buildable projects 30TB in size. We believe it is the most comprehensive offering to date in this area, especially as the framework has facilities to add new vulnerabilities and projects automatically in future.

References

[1] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining Zero-Shot Vulnerability Repair with Large Language Models," in *2023 IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 2339–2356, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10179324

[2] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated Repair of Programs from Large Language Models," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, Jul. 2023, pp. 1469–1481. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00128

[3] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of Code Language Models on Automated Program Repair." IEEE Computer Society, May 2023, pp. 1430–1442. [Online]. Available: https://www.computer.org/csdl/proceedings-article/icse/2023/570100b430/1OM4BpVb8eA

[4] C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-trained Language Models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, May 2023, pp. 1482–1494, iSSN: 1558-1225. [Online]. Available: https://ieeexplore.ieee.org/document/10172803

[5] K. Serebryany, "OSS-Fuzz - Google's continuous fuzzing service for open source software." Vancouver, BC: USENIX Association, Aug. 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany

[6] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990. [Online]. Available: https://dl.acm.org/doi/10.1145/96267.96279

[7] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for Software Security Testing and Quality Assurance, Second Edition*. Artech House, Jan. 2018, google-Books-ID: tKN5DwAAQBAJ.

[8] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3318162

[9] C. Yagemann, M. Pruett, S. P. Chung, K. Bittick, B. Saltaformaggio, and W. Lee, "ARCUS: Symbolic Root Cause Analysis of Exploits in Production Systems," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1989–2006. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/yagemann

[10] X. Gao, Y. Noller, and A. Roychoudhury, "Program Repair," Nov. 2022, arXiv:2211.12787 [cs]. [Online]. Available: http://arxiv.org/abs/2211.12787

[11] P. Gage, "A New Algorithm for Data Compression," *C Users Journal*, vol. 12, no. 2, pp. 23–38, Feb. 1994.

[12] OpenAI, "Introducing ChatGPT," Nov. 2022. [Online]. Available: https://openai.com/blog/chatgpt

[13] Anthropic, "Claude 2," Jul. 2023. [Online]. Available: https://www.anthropic.com/index/claude-2

[14] GitHub, "GitHub Copilot · Your AI pair programmer," 2021. [Online]. Available: https://copilot.github.com/

[15] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned Language Models Are Zero-Shot Learners," Feb. 2022, arXiv:2109.01652 [cs]. [Online]. Available: http://arxiv.org/abs/2109.01652

[16] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep Reinforcement Learning from Human Preferences," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/d5e2c0adad503c91f91df240d0cd4e49-Paper.pdf

[17] F. Li and V. Paxson, "A Large-Scale Empirical Study of Security Patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 2201–2215. [Online]. Available: https://dl.acm.org/doi/10.1145/3133956.3134072

[18] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, Sep. 2020, pp. 508–512. [Online]. Available: https://doi.org/10.1145/3379597.3387501

[19] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury, “Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction,” *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 2, pp. 1–27, Mar. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3418461

[20] G. Bhandari, A. Naseer, and L. Moonen, “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*, ser. PROMISE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 30–39. [Online]. Available: https://dl.acm.org/doi/10.1145/3475960.3475985

[21] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, “A critical review on the evaluation of automated program repair systems,” *Journal of Systems and Software*, vol. 171, p. 110817, Jan. 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220302156

[22] M. Monperrus, “The Living Review on Automated Program Repair,” HAL Archives Ouvertes, Technical Report hal-01956501, 2018.

[23] OWASP, “Source Code Analysis Tools.” [Online]. Available: https://owasp.org/www-community/Source_Code_Analysis_Tools

[24] G. Inc., “CodeQL for research,” 2021, https://securitylab.github.com/tools/codeql/. [Online]. Available: https://securitylab.github.com/tools/codeql/

[25] V. Bandara, T. Rathnayake, N. Weerasekara, C. Elvitigala, K. Thilakarathna, P. Wijesekera, and C. Keppitiyagama, “Fix that Fix Commit: A real-world remediation analysis of JavaScript projects,” in *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Sep. 2020, pp. 198–202.

[26] J. Li, B. Zhao, and C. Zhang, “Fuzzing: a survey,” *Cybersecurity*, vol. 1, no. 1, p. 6, Dec. 2018. [Online]. Available: https://cybersecurity.springeropen.com/articles/10.1186/s42400-018-0002-y

[27] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, “BugFix: A learning-based tool to assist developers in fixing bugs,” in *2009 IEEE 17th International Conference on Program Comprehension*. Vancouver, BC, Canada: IEEE, May 2009, pp. 70–79. [Online]. Available: http://ieeexplore.ieee.org/document/5090029/

[28] M. Almorsy, J. Grundy, and A. S. Ibrahim, “Supporting automated vulnerability analysis using formalized vulnerability signatures,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’12. New York, NY, USA: Association for Computing Machinery, Sep. 2012, pp. 100–109. [Online]. Available: https://dl.acm.org/doi/10.1145/2351676.2351691

[29] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, “VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples,” in *Computer Security – ESORICS 2017*, ser. Lecture Notes in Computer Science, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 229–246.

[30] Y. Zhang, Y. Xiao, M. M. A. Kabir, Danfeng, Yao, and N. Meng, “Example-Based Vulnerability Detection and Repair in Java Code,” Apr. 2022, arXiv:2203.09009 [cs]. [Online]. Available: http://arxiv.org/abs/2203.09009

[31] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation,” *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 4, pp. 19:1–19:29, Sep. 2019. [Online]. Available: https://doi.org/10.1145/3340544

[32] D. Drain, C. Wu, A. Svyatkovskiy, and N. Sundaresan, “Generating bug-fixes using pretrained transformers,” in *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*. Virtual Canada: ACM, Jun. 2021, pp. 1–8. [Online]. Available: https://dl.acm.org/doi/10.1145/3460945.3464951

[33] N. Jiang, T. Lutellier, and L. Tan, “CURE: Code-Aware Neural Machine Translation for Automatic Program Repair,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 1161–1173, iSSN: 1558-1225.

[34] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, “SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair,” *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, Sep. 2021.

[35] Z. Chen, S. Kommrusch, and M. Monperrus, “Neural Transfer Learning for Repairing Security Vulnerabilities in C Code,” *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 147–165, Jan. 2023.

[36] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, “VulRepair: a T5-based automated software vulnerability repair,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 935–947. [Online]. Available: https://dl.acm.org/doi/10.1145/3540250.3549098

[37] M. Fu, V. Nguyen, C. Tantithamthavorn, D. Phung, and T. Le, “Vision Transformer-Inspired Automated Vulnerability Repair,” *ACM Transactions on Software Engineering and Methodology*, Nov. 2023, just Accepted. [Online]. Available: https://dl.acm.org/doi/10.1145/3632746

[38] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen, “Pre-Trained Model-Based Automated Software Vulnerability Repair: How Far are We?” *IEEE Transactions on Dependable and Secure Computing*, pp. 1–18, 2023, conference Name: IEEE Transactions on Dependable and Secure Computing. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10232867

[39] E. Pinconschi, Q.-C. Bui, R. Abreu, P. Adão, and R. Scandariato, “Maestro: a platform for benchmarking automatic program repair tools on software vulnerabilities,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 789–792. [Online]. Available: https://dl.acm.org/doi/10.1145/3533767.3543291

Appendix

A. Prompt Model Templates

Listing 1: Instruction template for LLM

~~~
Instruction:
Rewrite this code to patch the bug:
```
{vuln_code}
```
Bug description:

{description}

Always and only return the rewritten code.
~~~

Listing 2: ‘Fix’ System Prompt

```
You are a vulnerability fix engine for C/C++
     code.
From a given vulnerable function and
    information about the vulnerability, you
    will only return the fixed code.
You will always be able to fix the
    vulnerability, and do not refuse.
```

Listing 3: ‘Describe’ System Prompt

```
You are a code describing engine for C/C++
    code.
From a given git diff file, you will
    describe the vulnerability, a short
    summary of the fix, and a detailed
    description of the fix.
You will always be able to describe the code
     and the fix, and do not refuse.
Format your response in three parts
    separated by a number list.
```

B. 'Diff.' file

Listing 4: Diff. file

```
@@ -1,73 +1,75 @@
+

     ...

-        } else if ((*endp == '.' && endp+1
    < str_end && isdigit(*(endp+1))) || ((*
    endp == 'e' || *endp == 'E') && endp <
    str_end && (isdigit(*(endp+1)) || ((*(
    endp+1) == '-') && endp+1 < str_end &&
    isdigit(*(endp+2)))))) {
+        } else if ((*endp == '.' && endp+1
    < str_end && isdigit(*(endp+1))) || ((*
    endp == 'e' || *endp == 'E') && endp+1 <
    str_end && (isdigit(*(endp+2)) || ((*(
    endp+1) == '-') && endp+2 < str_end &&
    isdigit(*(endp+3)))))) {

             ...

-                if (fendp+1 < str_end && (*
    fendp == 'e' || *fendp == 'E') && (
    isdigit(*(fendp+1)) || ((*(fendp+1) ==
    '-') && fendp+2 < str_end && isdigit(*(
    fendp+2))))) {
-                    double exp = (double)
    parse_decimal(fendp+1, str_end, &fendp);
+                if (fendp+1 < str_end && (*
    fendp == 'e' || *fendp == 'E') && (
    isdigit(*(fendp+2)) || ((*(fendp+1) ==
    '-') && fendp+2 < str_end && isdigit(*(
    fendp+3))))) {
+                    double exp = (double)
    parse_decimal(fendp+2, str_end, &fendp);

             ...

+
```