Examining Zero-Shot Vulnerability Repair with Large Language Models

Hammond Pearce*, Benjamin Tan†, Baleegh Ahmad*, Ramesh Karri*, Brendan Dolan-Gavitt*

*New York University, †University of Calgary

Abstract—Human developers can produce code with cybersecurity bugs. Can emerging ‘smart’ code completion tools help repair those bugs? In this work, we investigate the use of large language models (LLMs) for code (such as OpenAI’s Codex and AI21’s Jurassic J-1) for zero-shot vulnerability repair. We seek to characterize the feasibility of using black-box, “off-the-shelf” LLMs, as well as an open-source model and our own locally-trained model, on a mix of synthetic, hand-crafted, and real-world security bug scenarios. Our experiments demonstrate that while the approach has promise (the LLMs could collectively repair 100% of our synthetically generated and hand-crafted scenarios), a qualitative evaluation of the model’s performance over a corpus of historical real-world examples highlights challenges in generating functionally correct code.

Index Terms—Cybersecurity, AI, code generation, CWE

I. INTRODUCTION

Commercial large language models (LLMs), trained on vast amounts of source code, have been enthusiastically promoted as tools to help developers in general coding tasks like translating between programming languages and explaining code [1]–[4] by predicting likely text completions given some “prompt” comprising comments, function names, and other code elements. This is similar to the multi-tasking capabilities that LLMs for natural language exhibit [5], [6]. Of the many tasks coders do, we are interested in fixing security bugs; developers might ordinarily run security tools such as fuzzers or static analyzers, try to understand the feedback, locate the issue, and modify the code to repair the bug. This is hard.

In this paper, we ask: Can LLMs for code completion help us fix security bugs (Fig. 1)? “Out-of-the-box” LLMs for coding, such as OpenAI’s Codex [7] and AI21’s Jurassic J-1 [8] are trained on open-source code in myriad languages that contain a large variety of comments [9]–[11] and functionality (both buggy and non-buggy). This powers the ability of LLMs to complete code in different ways, given some context such as the designer’s intent in code comments.

While recent work [12] suggests that code completions with the LLM GitHub Copilot can introduce security weaknesses, Pearce et al. conclude that models can still “increase the productivity of software developers”, especially when paired with “appropriate security-aware tooling during...generation to minimize the risk” [12]. As one can guide LLMs by adding cues to prompts (as suggested by user guides like [4]), we seek to characterize the feasibility of using black-box, “off-the-shelf” LLMs for “zero-shot” generation of replacement code for an identified security bug, perhaps as part of an overarching program repair framework. This contrasts prior work that trained specialized neural machine translation (NMT)-based models for fixing software bugs (e.g., [13]–[16]). Unlike prior approaches which are trained to predict the exact same tokens as a human developer’s fix, we want to investigate off-the-shelf LLMs’ apparent ability to “understand” the broader context from a source code file.

We focus on creating security patches as they are important, can be tricky to write, and often require relatively small changes of code in a single file [17]. The smaller footprint of security patches suggests that current off-the-shelf LLMs might be capable of designing bug fixes. We want to know if LLMs—despite not being trained specifically on security fixes (and, indeed, being trained on a great deal of insecure code)—are nonetheless capable of generating valid fixes for vulnerable code. We seek answers to these research questions:

RQ1: Can off-the-shelf LLMs generate safe and functional code to fix security vulnerabilities?
RQ2: Does varying the amount of context in the comments of a prompt affect the LLM’s ability to suggest fixes?
RQ3: What are the challenges when using LLMs to fix vulnerabilities in the real world?
RQ4: How reliable are LLMs at generating repairs?

To answer these questions, we evaluate recent LLMs on a range of synthetic, hand-crafted, and real-world buggy scenarios. Our contributions are as follows.

1) We compare prompts, contextual cues, and model settings (temperature, sampling strategy, etc.) for encouraging LLMs to generate functional and secure code. (Section III)
2) We provide the first evaluation of LLMs for zero-shot generation of security fixes, showing that off-the-shelf models are capable of producing security fixes without any additional

1From this point on, our discussion of LLMs is specifically about off-the-shelf, general coding LLMs available at the time of this study.
training in simple scenarios (Section IV). However, when we evaluate the LLMs on some real-world scenarios (Section V), we find that they can struggle to generate plausible fixes and as such are still not ready to provide real-world value in a program repair framework. (3) To encourage further research into the use of LLMs in vulnerability repair, we open-source our datasets and evaluation framework, including the training data and trained model for ‘gpt2-csrc’.

II. BACKGROUND AND MOTIVATION

A. Security Bugs

Developers perform many tasks in creating and maintaining software; they can sometimes fall into patterns of insecure code, such as those in MITRE’s Common Weakness Enumeration (CWE) database [18]. Challenges around security bugs include detecting them, localizing root causes, understanding root causes, and creating/testing patches.

Several tools and techniques try to identify security bugs statically, such as those in OWASP’s list [19]. Developers can also use run-time sanitizers like Address Sanitizer (ASAN) [20] and Undefined Behavior Sanitizer (UBSAN) [21] to identify a bug’s root cause. Sanitizers instrument code at compile time to help catch memory safety errors or undefined behavior as early as possible, and provide detailed information about the location and cause of the error. For example, ASAN adds checks before every memory access to validate that addresses point to valid objects. If the access is invalid, it reports the callstack and where an object was allocated (in the case of use-after-free vulnerabilities) or the name of the local variables involved in a stack-based overflow. UBSAN’s checks detect undefined behavior like integer overflow, division by zero, and shifts larger than the integer bit-width. Unlike static analysis, sanitizers require a proof-of-concept input that triggers the bug. For security bugs, particularly those found by fuzzers, such inputs are often included with the bug report.

Given a bug report, what should a developer do? To date, expert developers develop patches by hand [17], with ongoing research efforts toward automatic program repair [22]. Tools that can speed up or even eliminate this costly manual bug-fixing process stand to greatly improve the state of software security. In this work, we want to use LLMs to repair identified security bugs by generating alternative replacements.

B. “Prompting” Large Language Models (LLMs)

Broadly, LLMs act as ‘Scalable sequence prediction models’ [1]: given a prompt comprising a sequence of tokens, they output the ‘most likely’ set of tokens that continue/complete the sequence (similar to a smart autocomplete). For example, an LLM can be asked to complete a function body, given some signature and/or comment [1].

Here, tokens refer to common character sequences. Most LLMs (including those evaluated in this work) operate on tokens, not individual characters. Each token has a unique numeric identifier, up to a user-defined vocabulary size. This byte pair encoding (BPE) [23] process allows the models to ingest more text into their (fixed-size) input window. For Codex, which uses the same tokenizer as GPT-3 (extended to include tokens for runs of whitespace to better deal with code indentation) [1], an average token represents ≈four characters.

Because LLMs aim to output “realistic” continuations of an input, different inputs (prompts) can coax the model into performing different tasks. LLM responses to individual prompts can be tuned further using parameters such as temperature (roughly, the model’s propensity for choosing unlikely tokens), top-p (consider only tokens up to a cumulative probability p), length (how many tokens to generate), and stop words. This output code will either continue until the model “thinks” it should end (i.e., ‘stopping’ was the most-likely next token), or until it reaches a pre-specified length.

C. Studied Off-the-Shelf Large Language Models

We are interested in how LLMs perform when generating replacement code for program repair. We evaluate several LLMs, summarizing their essential characteristics in Table I. These include OpenAI’s Codex models [1], AL21’s Jurassic-1 models [8], [25], and the ‘polycoder’ model [26]. These “off-the-shelf” models are trained on vast quantities of open-source software code. In Codex’s case, this includes the majority GitHub’s open-source code. As a result of both the difficulty involved in their training, their size (billions of parameters), and potential commercial importance, these LLMs are ‘blackbox’. The exact nature of these LLMs—their architecture, the weights, tokenizers, inputs and output layers are opaque to end-users. Even if they were ‘open’, these LLMs are currently infeasible to run on local machines due to their mammoth size and complexity. Instead, they are accessed via managed internet APIs. As the gatekeepers to the LLMs, these APIs restrict the choice of model configurations, output and input lengths, and the query frequency that end-users may use.

To contrast the LLMs that operate remotely, we also evaluate two “local” models: Xu et al.’s recent ‘polycoder’ model [26] and our own locally-trained C/C++ code model, ‘gpt2-csrc’. With 2.7 billion and 774 million parameters, respectively, ‘polycoder’ and ‘gpt2-csrc’ are small enough to run locally on a high-end consumer GPU.

2 A full discussion of what parameters influence LLM generation is beyond the scope of this paper, but Huggingface’s tutorial on decoding methods for Transformers [24] is a good introduction.

3 The exact amount is not published, although GitHub Copilot—the commercial adaptation of Codex—advertises itself as being trained over ‘billions’ of lines of code [2].
While prior work examined GitHub Copilot’s performance in security-relevant contexts [12], we exclude Copilot in this study for three reasons: (1) at the time of this study, access is restricted via a wait-list, (2) once past the wait-list, access to the model suggestions is restricted to closed-source, graphically-based integrated development environment plugins that are not suitable for large-scale studies, and (3) given that Copilot is based upon Codex, we feel that evaluating Codex will likely exhibit comparable performance.

D. Design of `gpt2-csrc`

Although we expect that the larger, commercial LLMs will be more effective at vulnerability repair, including our locally trained LLM has practical and scientific benefits. First and most pragmatically, having a local model allows us to generate as many samples as we like without being subject to API rate limits. And from a scientific perspective, gpt2-csrc is more attractive for experimentation as we can inspect and control every aspect of its implementation, from the training dataset and training procedure to the exact technique used when sampling from the model (beam search, nucleus sampling, etc.). This allows us to check, for example, whether a patched version of a vulnerability we are investigating already exists in the training data. Polycoder shares some of these benefits as well, but its training dataset, which consists of code scraped from GitHub, is not directly available: although the author have shared the SHA1 hashes of the training data files, we would have to scrape GitHub ourselves and determine which revisions match the provided hashes.

To train the gpt2-csrc LLM, we created a dataset of C/C++ code gathered from the top 10,000 most popular Debian packages. We preprocessed the training data to filter out non-source code files (using common filename extensions used for C/C++ code, such as `.c`, `.cpp`, `.h`, etc.), removed files that were not plain-text, and deduplicated the source files using locality-sensitive hashing, resulting in ≈17GB of code. For tokenization we trained a BPE tokenizer on the source dataset. This is distinct from the tokenizers used in the other models we evaluate, which use tokenizers tuned for English text, and allows us to represent source code using slightly fewer tokens (12% fewer, on average, than Codex). Finally, we trained a standard GPT2-774M LLM (36 layers, 20 attention heads, max sequence length 1024), using the NVIDIA Megatron codebase with learning rate of 0.0001, weight decay of 0.01, and a cosine learning rate decay schedule. The model was trained for four GPU-months (one month on 4×RTX8000 GPUs).

III. FROM PROMPTS TO PROMPT ENGINEERING

LLMs trained on large corpora of data can inadvertently gain some ability to perform multiple tasks, even in a “zero-shot” setting [5], [6]. Similarly, LLMs for code can perform several tasks (e.g., [27]) by careful construction of the “prompt” (the sequence of tokens we provide to the model). Because predicted tokens are based on probabilities, there is no guarantee that a given prompt will make the model do what a user intends (i.e., alignment failure [1])—we can only hope that prompts coax a model towards what we intend.

To date, the notion of “prompt engineering” is nascent, with empirical prior work showing that the security code weaknesses in a model’s (GitHub’s Copilot) outputs are sensitive to the prompt [12], which comprises both existing code and comments in a given snippet to be completed. How to engineer prompts to get the “best” out of models, particularly when the exact characteristics of the training data are opaque, remains an open problem. Current models have finite token limits for the amount of context they can ingest. Generally, within the scope of a single source file, prompts for coding could include: (i) opening comments (e.g., notices, change histories) (ii) imports and defines (e.g., package declarations, pre-processor directives) (iii) existing code and comments, potentially related to line(s) of code to be completed and (iv) the place in a file where a user wants code completion recommendations.

If a source file is small enough for a model to ingest, the entire file can be the prompt; if not, we need to be judicious in selecting the parts used to assemble the prompt (the practical implications of this become apparent in Section V). If we want a specific type of recommendation from a model, we need to modify the prompt appropriately, e.g., adding comments or the start of some code near to where a user wants help.

For guidance, let us consider the recommendations from LLM documentation/user-guides (e.g., [4]). Prompts should “start with a comment, data or code” and contain information that might be helpful or needed by a “programmer…to perform a task”. Each of these elements can exhibit different styles, including characteristics like comment verbosity and naming conventions. Prior studies of code comments in open-source software reveal a wide range of reasons for them [9]–[11], [28], including marking intent for developers, such as explaining functionality or indicating where fixes should occur (e.g., marking something as FIXME or FIXED). We observed the existence of “BUG” and “FIXED” in comments from searching GitHub and adopted those as potential elements of a prompt. Similarly, source code can appear inside comments; “commented-out” code often appears in code commits as part of the bug-fixing process [29]. Comments vary in complexity, from sparse single line comments that describe something as a “bugfix” through to verbose comments that include prior

---

4According to the Debian Popularity Contest, https://popcon.debian.org/.

---

| Bug ID | Program | Security Tests | Bug Repair |
|-------|---------|----------------|------------|
| 2341  | Bug     | ORM            | Fix        |
|       |         | Loc                          |            |

Fig. 2. We focus on understanding LLMs’ generation of replacement code that “repairs” a security bug (shaded area). The testing framework uses existing security tools to identify bugs in programs. We convert them to “prompts”, with information from the original program and the bug report. LLMs ingest “prompts” to produce potential repaired code. Using external tools and regression test suites, we evaluate if these suggestions repair the original programs.
instances of code (thus providing a high level of context). We explore comment types for prompting LLMs in Section IV.

IV. RQ1 / RQ2: SYNTHETIC EXPERIMENTATION

A. Overview

To begin measuring how well LLMs can help with ‘fixing security bugs’ we generate a large, synthetic collection of buggy programs (Section IV-B), and attempt repair of these with Codex LLMs with different parameter settings (Section IV-C). We then broaden to the other LLMs and domains using handcrafted vulnerable programs and prompts (Section IV-D and Section IV-E).

The first stage of this experiment is a Model Parameter Sweep to identify good ‘typical’ parameters for LLM code repair generation, and then the second stage investigates prompts to determine if different prompt patterns with increasing ‘bug context’ has an impact on the quality of code generated by black-box LLMs. For our experiments, we designed and implemented the framework depicted in Fig. 2. While this resembles an automated repair system, its function is to automate and scale the investigation for investigating LLM performance. As a base case, we first provide a vulnerable program (scenario) to a security evaluation tool which generates a bug report; we adopted CodeQL [30] given its ability to statically analyze several types of bugs. From the bug report and the original program, we derive a prompt which is provided to the LLM. This returns a code completion which can be merged with the original program to provide a potentially fixed program. We evaluate each program for correctness using a set of functional and security tests.

Experimental Platform: We perform our LLM experiments on a single desktop-class PC with Intel i7-10750H processor, 16 GB DDR4 RAM, NVIDIA RTX 2070 with 8 GB VRAM; using Ubuntu 20.04. We develop our framework using Python 3.8.10 and gcc 9.3.0-17. We use CodeQL version 2.7.2. Open-source: all code and the generated data is available. See the Appendix / Ref [31].

B. Model Parameter Sweep: Vulnerable Program Generation

In this section we check the effect of model parameters (specifically, temperature and top_p) on LLMs’ generated code. Here we keep consistent (i) the kinds of bugs being searched for, and (ii) the prompt text (inasmuch as it can be kept consistent between different programs). This means that we need many kinds of examples of the same bug. For this exercise we draw on prior work [12] that demonstrated LLMs’ propensity for generating vulnerable code and use OpenAI’s Codex to generate vulnerable programs.

CWE choice We experiment on two synthetic examples for two notable CWEs—CWE-787: Out of bounds Write (Rank #1 on MITRE’s “Top 25 List” [32]); and CWE-89: Improper Neutralization of Special Elements used in an SQL Command (‘SQL Injection’) (Rank #6). We select these two CWEs because (1) as noted by MITRE, they are high-impact bugs. Their presence in an application can lead to catastrophic consequences, including data loss and exfiltration and root privilege escalation; (2) they are ‘concrete’ in that their presence or absence can be determined directly from the given code, that is, they do not require any additional context; and (3), they are from two different levels of abstraction. Memory buffer errors tend occur in “lower-level” languages like C, and SQL injection errors tend to be in “higher-level” languages like Python, meaning we gain a sense of the LLM performance across two different aspects of software development.

Synthetic generation To generate large numbers of unique but similar vulnerable programs, we specify (i) the beginning of a short program relevant to these CWEs, (ii) get the target LLM to complete the program, and then (iii) evaluate those programs for functional and security correctness by running them through unit tests and CodeQL, respectively. We then (iv) take the set of unique functional but vulnerable programs as our dataset for bug-fixing. The initial program prompts are listed in Fig. 3(a), which is a simple C program for multiplying two floats and returning the result as a string; and Fig. 3(b), which is a small Python component that used to unsubscribe emails from a mailing list in a SQL database.

From these two programs we prompted the OpenAI Codex engines ‘code-cushman-001’ and ‘code-davinci-001’ to generate 10 programs for each combination of temperature \( \{0.00, 0.25, 0.50, 0.75, 1.00\} \) and top_p \( \{0.00, 0.25, 0.50, 0.75, 1.00\} \). This resulted in 250 suggestions for each model, or 500 overall. We then try to compile each program to check if it is valid (i.e., that it can be compiled) before checking for functional and security correctness. For interest, we include the statistics for the initial generation in Table VIII in the Appendix.

In total the two language models generated 95 unique buggy programs containing CWE-787 (where the suggested code does not allocate enough memory for the string returned by multiply_doubles_str, which, if filled using sprintf, needs to be at least 318 bytes, as the maximum length of a double represented as a string is 317 characters plus a null byte), and 22 unique buggy programs containing CWE-89 (where the SQL query assembled in the function unsubscribe_email_from_all_subscriptions are vulnerable to SQL injection—i.e., if the string was constructed using concatenation). Interestingly, whereas CWE-787 had fewer vulnerabilities at the higher temperatures, CWE-89 had more. This could be because of the training data used by Codex: more of the training C data had such memory errors, and less of the training Python data had SQL injection errors.

C. Model Parameter Sweep: Vulnerable Program Repair

We now take the unique functional and vulnerable programs from CWE-787 (95 of the Functional and Vulnerable programs) and CWE-89 (22 of the Functional and Vulnerable programs) and augment the original prompt with the faulty code identified by CodeQL to produce new repair scenarios. Here, each faulty program is augmented in the same manner. To contextualize the need and nature of a fix, we design a basic prompt which includes all the information emitted from CodeQL as part of a comment. We also comment out the
entire function’s worth of vulnerable code. It is likely, however, that the LLM would present different results given a different prompt design. For instance, if no context was provided, then the LLM results would likely appear as they did in the original prompt design. For instance, if no context was provided, then the LLM results would likely appear as they did in the original prompt design.

An example repair prompt from CWE-787 is listed in Fig. 4. Here, the original vulnerable code (as generated by Codex, commented-out in the prompt) stores the string in a malloc-ed variable of size 16 (sizeof(char) * DBL_DIG + 1 = 1 * 15 + 1), when instead, it needs a much longer 318-bytes (or greater) string variable. We query the LLMs with each of these repair prompts, again performing a sweep of the parameters across the temperature and top_p range used prior. This results in 95 x 500 = 47500 programs for the scenarios based on CWE-787, and similarly, 11000 programs for CWE-89. The results of these unique (but similar) programs are combined and summarized in Table II.

Results: Table II summarizes the synthetic program repair results. 2.2% of the 22034 valid programs for the CWE-787 scenario were repaired. 29.6% of the 10796 valid programs for the CWE-89 scenario were repaired.

Our Observations: From these two experiments (Table II) we can observe that the (Codex) LLMs are capable of repairing vulnerable programs (RQ1) when given a suitable repair prompt. CWE-787 scenario saw 491 suggested patches that fix the bug (2.2% of suggestions), and CWE-89 saw 3197 (29.6%). In addition, given that the original programs were
also generated by the language model without bugs, it is also worth noting that the LLMs are capable of generating bug-free code even without additional context—assuming that they ‘understand’ the generation task (i.e., it is simple, like the original prompts in Fig. 3). Indeed, the original generation task for the two scenarios gave fewer buggy programs (4.4% of the programs were bug-free for CWE-787; 93.6% for CWE-89) than the aggregated repair prompts (2.2% and 29.6% of patches were bug-free respectively). This indicates that while both scenarios were repaired by this prompt, the contents of the repair prompt should be further investigated (RQ2). Further, given the results presented in Fig. 5, where higher temperatures are better for CWE-787 and worse for CWE-89 and vice versa, we can conclude that there is no single temperature/top_p that will best cover all scenarios and engines (perhaps to be expected). However, given that the official Codex documentation advises varying either one of these parameters but not both, we can choose to set as parameters for an ensemble of queries a set of temperatures \{0.00, 0.25, 0.50, 0.75, 1.00\} with a top_p fixed to 1.00. This prunes the “search space” for potential repairs by 80% in our subsequent experiments.

### D. Prompt Engineering and Hand-Crafted Vulnerable Code

For our next experiments, we dig deeper into RQ2, by (i) increasing the variety of prompts (increasing and decreasing the amount of context in the comments); and (ii) examining a wider and more complex range of scenarios. Further, instead of generating scenarios synthetically via Codex (which may have follow-on implications upon the generated code) we hand-write short bespoke programs containing security weaknesses from MITRE’s “Top 25” [32] list. As an additional point of comparison, we expand our analysis to include other LLMs—the polycoder and gpt2-csrc models, which we ran locally; and AI21’s ‘j1-large’ and ‘j1-jumbo’ (although unfortunately, due to their comparatively lower API usage limit, we were forced to reduce our sampling for these by 50%).

#### CWE choice and scenario design

Experimental scenarios are summarized in Table III. They are designed from a selection of CWEs chosen for reasons similar to those in Section IV-B: they are high-impact (in the ‘Top 25’ list by MITRE [32]), concrete (straightforward to test presence/absence), and varied, with “higher-level” bugs inspired by Python web development and “lower-level” bugs being C buffer and pointer scenarios. These CWEs are a subset of those analyzed by automated tools in [12], and our scenarios are inspired from their prior work.

#### Repair prompt design

After analysis by CodeQL, which finds our deliberately-inserted bugs, our experimental framework augments each scenario with a number of different possible prompt comment templates to form the repair prompts (Fig. 2). As there are many possible wordings and language choices for constructing a repair prompt, we use guidance from user guides, informal searches of GitHub, and existing literature (see Section III) to design five reasonable templates. These templates vary the amount of context provided to the language model (i.e., in C commenting, // and /* rather than //)).

### Table III: The different hand-crafted vulnerable scenarios.

| Scenario (‘Top 25’ Rank) | Lang. | Description | Weakness | LoC |
|-------------------------|-------|-------------|----------|-----|
| CWE-79 (#2)             | Python| A Flask blueprint for a web component to construct a simple welcome page. | Does not protect against cross-site scripting. | 16  |
| CWE-125 (#3)            | C     | A function which should return a value from an array of integers or -1 if the index is out of bounds. | Does not check the lower bound. | 20  |
| CWE-20 (#4)             | Python| A Flask blueprint to redirect users to a given destination URL on the ‘example.com’ domain. | Vulnerable to a prefix-domain name attack (e.g. redirect to ‘compromised-example.com’). | 14  |
| CWE-416 (#7)            | C     | Perform an XOR encryption of two strings. | A use-after-free bug. | 49  |
| CWE-476 (#15)           | C     | A program to calculate the value of a person’s account given assets and debts. | A null pointer exception when person name is unsuitable. | 42  |
| CWE-119 (#17)           | C     | A program which, given an ID number, returns a vehicle ID for that number from an array. | Does not check any bounds. | 15  |
| CWE-732 (#22)           | C     | A program which writes a secret to a protected file. | Does not set appropriate permissions on the file when creating. | 23  |

### Table IV: Templates for repair prompt generation.

| Template ID | Description |
|-------------|-------------|
| n.h.        | No Help - deletes the vulnerable code/function body and provides no additional context for regeneration. |
| s.1         | Simple 1 - deletes the vulnerable code/function body and adds a comment ‘bugfix: fixed [error name]’. |
| s.2         | Simple 2 - deletes the vulnerable code/function body and adds a comment ‘fixed [error name]’ bug. |
| c.          | Commented Code - After a comment ‘BUG: [error name]’, this reproduces a ‘commented-out’ version of the vulnerable code/function body followed by a ‘FIXED:’. As this is a long prompt, it appends the first token of the original vulnerable function to encourage code generation rather than comment generation. |
| c.m.        | Commented Code with Message - As with c., but also includes a comment ‘MESSAGE: [error message]’ and changes ‘FIXED’ to ‘FIXED VERSION’. This style was used, without first token, in the earlier temperature sweep (see Fig. 4). |
| c.a.        | Commented Code (alternative - Used for real-world examples, see Section V). As with c., but commented in the alternative style for C programs (i.e., in C commenting, // and /* rather than /*/). |
| c.n.        | Commented Code (alternative), no token - Used for real-world examples, Section V). As with c.a., but with no ‘first token’ from vulnerable code. |
```python
def hello(username):
    return "Hello, %s!" % username
```

(b) Template n.h. returns a safe program that fails safety tests, as the prompt (lines 1-3) does not indicate (`<p>` / `</p>`) tags are required.

```python
def hello(username):
    # BUG: Reflected server-side cross-site scripting
    return "Hello, %s!" % username
    # FIXED:
    return "Hello, %s!" % escape(username)
```

c. Template c. returns a safe and functional program.

```
Similar to Fig. 7. CWE-79 (Python component with XSS vulnerability) program repair. Highlighted code was generated by the LLMs.
```

### Table V

| Template ID | # Gen. | # Vld. | # Vuln. | # Fn. & Vuln. | # Fn. & Safe | % Vld. | Repair |
|-------------|--------|--------|---------|---------------|--------------|--------|--------|
| n.h.        | 2000   | 1316   | 340     | 646           | 116          | 530    | 40.2 |
| s.1         | 2000   | 1213   | 247     | 539           | 94           | 445    | 36.7 |
| s.2         | 2000   | 1204   | 315     | 592           | 126          | 466    | 38.7 |
| c.          | 2000   | 1345   | 561     | 1140          | 475          | 665    | 49.4 |
| c.m.        | 2000   | 1315   | 478     | 1104          | 414          | 690    | 52.5 |

Gen. (Generated), Vld. (comparable), Vuln. (Vulnerable), Fn. (Functional), Safe (Not Vulnerable)

The performance of each template presented collectively in Table V and the total number of functional and safe programs generated by each engine presented in Fig. 14(a) (in the Appendix).

**Our Observations:** It is difficult to draw a definitive conclusion: the performance widely varied between prompt, scenario, and LLM. However, any one correct code completion is all that is required to fix a given bug, and all scenarios were successfully repaired by at least one combination of template and engine. The performance of each individual template can guide future work towards making a robust general-purpose single prompt (RQ2). In some cases, such as CWE-20, the low-context templates (n.h., s.1, and s.2) out-perform the high-context templates (c. and c.m.), matching the results from the previous section (where original program generation performed better than program repair). However, in some scenarios, such as CWE-79 and CWE-732, this is reversed, and the high-context templates perform better. When we perform a qualitative analysis of the results, we begin to understand this discrepancy. For many ‘low-context’ templates, the generated prompt may not provide enough information to successfully generate code that will pass functional testing. That is, the intent/functional of the removed code is not adequately explained by what remains. As such, even though the generated code may pass security tests, it fails to pass...
functional tests. This is the case with CWE-79, for instance, which is reproduced in Fig. 7(a). With a low-context template, the LLM is often unable to determine the output format for the ‘hello’ message without the context the commented out code provides (see Fig. 7(b)). This is because once the faulty line is removed, there is no information provided to describe what the output should look like as the comment describing the function is too vague. However, when the additional context is provided (e.g. via high-context templates c. and c.m.), where the vulnerable code is instead commented out (see Fig. 7(c)), more information is made available to the LLM, and now outputs may be generated correctly. This helps to explain the aggregated results in Table V, where the high-context templates produced the best results on average. We believe that the additional technical detail provided helps LLMs to generate code to pass both security and functional tests for more difficult scenarios while still being useful for simpler bug-fixes. As such, we believe this indicates that a more robust prompt should include more details rather than fewer.

The OpenAI Codex models consistently outperform the other models with regards to generating successful patches. Given the apparent value in verbose prompts, we hypothesize that this relative performance is due to the broad training data in the original Codex models (i.e., based on GPT-3 models which were trained over English text, potentially leading to a better ‘understanding’ of the comments). Nevertheless, all models, including our own ‘gpt2-csrc’ were able to repair at least some of the programs.

E. Repairing Hardware CWEs

As another component of our initial characterization of LLMs, we consider a more esoteric code completion setting: code written in hardware design languages such as Verilog (which was explored in prior work [12]). We now evaluate LLMs’ ability to fix scenarios involving hardware CWEs. Buggy Verilog code may be used to understand the confidentiality and/or integrity of the designed hardware.

CWE choice and scenario design: To examine the LLM performance on hardware, we designed two buggy hardware scenarios into our experimental data set. These scenarios are based on two Hardware CWEs which were chosen due to their relative straightforwardness: like the software CWEs examined above, their presence or absence within a snippet of Verilog code can be relatively easily determined. In addition, the code that is generated is relatively simple, which is important for these LLMs that have not been trained on extensive quantities of Verilog code. For similar reasons, these CWEs were also among those examined by the authors in [12]. The first hardware CWE we examine is CWE 1271: Uninitialized Value on Reset for Registers Holding Security Settings. This arises when security-relevant assets such as locked registers are not set to appropriate values upon reset. The second is CWE 1234: Hardware Internal or Debug Modes Allow Override of Locks, which arises when a security-relevant asset in a locked register which should only be modifiable (when it is unlocked) is accessible or can be modified during debug mode.

| Scenario, Engine | Prompt Template |
|------------------|----------------|
| code-cushman-001 | 231, 231 | 190, 259 | 13, 223 | 3, 246 | 3, 246 | 0.245 |
| code-davinci-001 | 228, 228 | 221, 229 | 29, 249 | 9, 223 | 8, 249 | 0.245 |
| code-davinci-002 | 1, 1 | 0, 1 | 0, 2 | 225, 225 | 0.245 |
| j1-large         | 0, 13 | 1, 12 | 1 | 6 | 0, 0.21 | 0, 19 |
| code-davinci-001 | 210, 210 | 212, 212 | 173, 243 | 7, 243 | 24, 243 | 6, 242 |
| code-davinci-002 | 225, 225 | 21, 245 | 8, 235 | 5, 247 | 22, 235 |
| code-davinci-002 | 224, 224 | 24, 249 | 22, 248 | 29, 245 | 22, 248 |
| j1-large         | 0, 0 | 1 | 0 | 0 | 0 | 0 |

Fig. 8. Results for LLMs when tasked with repairing two scenarios derived from the ‘Diversity of Domain’ dataset in [12]. For each engine/prompt, (1) 10 of each combination of temperature {0.00, 0.25, 0.50, 0.75, 1.00} and top_p {0.00, 0.25, 0.50, 0.75, 1.00}, giving 250 possible total programs; (5) of each combination of temperature {0.00, 0.25, 0.50, 0.75, 1.00} and top_p of 1.00 for AL21’s ‘j1-large’, giving 25 possible programs), (2) the results are presented as ‘safe and functional’/’valid (synthesizing) programs’.

To evaluate these CWEs, we adapt code from examples on the MITRE website [33]. Our scenario for CWE-1271 is 14 lines of Verilog code and 17 lines for CWE-1234. We pass these source files to Verilator [34] for functional and security testing before using these outputs within the framework (Fig. 2) with the prompt templates from Table IV.

Results: Results are presented in Fig. 8.

Our Observations: Empirically, we found that LLMs were less proficient at producing Verilog code than they were at C or Python, so we include a complete sweep of temperatures and top_p for the Codex models. Due to API usage restrictions we did not sweep the Jurassic models. We also exclude ‘polycoder’ and ‘gpt2-csrc’ as they do not support Verilog.

To increase the LLM code generation success, we add a post-processing step to Verilog generation. This is based around simple string analysis to add missing/remove redundant end and endmodule keywords. For functional testing, we simulate the generated modules with a range of inputs using Verilator testbenches for each scenario. For CWE-1271, this involves testing a lock register is locked on reset. For CWE-1234, we test that a lock register properly controls access to a data register. An example of correct and incorrect program synthesis is provided in Fig. 9. Here, the incorrect design has a bug whereby the lock register can be overwritten when the debug_unlocked or scan_mode signals are high. The fix involves removing this logic, such that the register no longer has any dependency upon them.

The hardware repair examples returned somewhat different behavior as compared to the software repair. As seen in Fig. 8,
the LLMs seemed perform better with less context provided in the prompt. In some sense, this can be thought of as the models having a tendency to ‘do the right thing.’ As these two hardware components are conceptually simple with straightforward specifications, it may be that having any errors in the design files is simply less probable than having bug-free versions.

V. RQ3: EXPERIMENTS WITH REAL-WORLD CVEs

A. Overview

To further understand the capabilities (and limits) of LLMs for code repair, we now investigate real-world scenarios (CVEs) from public open-source projects (RQ3). This introduces several benefits and challenges. Primarily, it allows us to characterize LLM performance for much larger and much more realistic software projects. The key challenge here is that we can no longer provide ‘full’ context in a source file to the LLMs—whereas previously the entire vulnerable program could fit in the token limit of each LLM, real programs tend to be much larger than the models can ingest. As such, we have to introduce a pre-generation reduction step to fit the prompt into the token limit.

B. ExtractFix Dataset

We collected 12 real-world vulnerabilities across three projects drawn from the ExtractFix dataset used in prior work on program repair [35]. We chose the vulnerabilities based on (i) whether we could find a proof-of-concept input that triggered the vulnerability; (ii) whether the developer-provided patch was localized to a single file (since existing language models generally cannot consider multiple files’ worth of context at once); and (iii) whether the project had a reasonably comprehensive test suite (for example, although ExtractFix contains two vulnerabilities in Jasper, a JPEG-2000 parser, Jasper did not have a test suite at the time these vulnerabilities were discovered). We note that restricting our evaluation to short, localized patches does not necessarily harm the validity of our analysis, as prior work has found that security patches tend to be more localized, have fewer source code modifications, and tend to affect fewer functions, compared to non-security bugs [17].

| Program | Description | File       | LoC | EF#  | CVE      |
|---------|-------------|------------|-----|------|----------|
| Libtiff | C library for processing TIFF files. | tiffcrop.c | 9.1k | EF01  | 2016-5321 |
|         |             | thumbnail.c | 683  | EF02-1 | 2014-8128 |
|         |             | tiff2pdf.c | 1.6k | EF02-2 | 2014-8128 |
|         |             | tif_pjpeg.c | 2.3k | EF08  | 2017-7601 |
|         |             | rgb2ycbcr.c | 393  | EF09  | 2016-3623 |
|         |             | tif_pjpeg.c | 2.4k | EF10  | 2017-7595 |
| Libxml2 | XML C parser and toolkit. | parser.c | 15.7k | EF15 | 2016-1838 |
|         |             | valid.c   | 15.4k | EF17 | 2012-5134 |
|         |             | jpeg.c    | 7k   | EF18  | 2017-5969 |
| Libjpeg-turbo | JPEG C library for manipulating JPEG files. | wrbmp.c | 557  | EF20 | 2018-19664 |
|         |             | jndmarker.c | 1.3k | EF22 | 2012-2806 |

Table VI provides an overview of the real-world projects and associated vulnerabilities that we tried to fix. The number of lines of code (LoC) varies even when the file to be fixed is the same, as each CVE corresponds to a different version (commit) of the project.

Bugs Localizing: Given our focus on characterizing LLMs in the zero-shot bug-fix setting, we used the developer-provided patches as an oracle to localize each vulnerability, prompting the LLMs to generate their fixes at the place in the code where the original developers patched each flaw. We note that although root cause identification and localization for security vulnerabilities is an active area of research [36]–[41], this question is orthogonal to the repair-based research questions we investigate in this work. Should LLMs prove useful for vulnerability repair, such work could be used as part of a complete, automated, end-to-end vulnerability repair system.

C. Code Reduction and Suggestion Consolidation

The token limits for each model, noted in Table I, functionally constrain the amount of code that (i) can be provided to the model, and (ii) can then be generated by the model (i.e., the number of tokens reflects both the input prompt and the output generated text). While these lengths were not an issue for the relatively short and simple programs in the handcrafted examples from Table III, the real-world scenarios in Table VI are too large to be presented in their entirety.

To resolve this we extend the experimental framework in Fig. 2 to support code files larger than the token limit. This extension consists of two parts, and is depicted in Fig. 10. The first reduces the amount of code provided to the language.
A. Bugs and Patches

(a) Buggy code from the tiffcrop.c file. Error line 9 is shown in red.

```c
for (row = 0; row < imagelength; row += t1)
{
    tbytes = TIFFReadTile(in, srcbuffs[s], col, row, 0, s);
    if (tbytes < 0 & & ignore)
    {
        TIFFError("readSeparateTilesIntoBuffer", "Error, can’t read tile
        for row_%lu_col_%lu."
    }
} /* END BUG FIX */
```

(b) Prompt constructed according to Fig. 10 (shortened for brevity). The red highlighted line 7 is the original faulty line indicated by ASAN/oracle. The template includes lines 8 and 9 (highlighted in grey) to encourage the LLMs to regenerate the safe code so the patch can be matched safely.

```c
for (s = 0; s < spp; s++)
{
    tbytes = TIFFReadTile(in, srcbuffs[s], col, row, 0, s);
    if (tbytes < 0 & & ignore)
    {
        TIFFError("readSeparateTilesIntoBuffer", "Error, can’t read tile
        for row_%lu_col_%lu."
    }
} /* END BUG FIX */
```

(c) Suggestion by code-cushman-001 which continues the prompt with the patch highlighted as yellow. Line 1 is the original file was safe.

```c
for (s = 0; s < spp; s++)
{
    tbytes = TIFFReadTile(in, srcbuffs[s], col, row, 0, s);
    if (tbytes < (tsize - ) (tw * nrow * bytes_per_sample))
    {
        TIFFError("readSeparateTilesIntoBuffer", "Error, can’t read tile
        for row_%lu_col_%lu."
    }
} /* END BUG FIX */
```

(d) The repaired program once reassembled with the LLM patched line 11 highlighted in yellow. This generated patch is semantically equivalent with the real-world human patched to repair this bug.

B. Findings

Results: The results for all prompts across all CVEs and LLMs are presented in Fig. 12. An LLM was able to repair 8 / 12 selected projects. Note that we define a project ‘repaired’ when the replaced code results in a compiled program passes both the functional tests included with each program and when it no longer crashes with the ASAN/UBSAN triggering input.

Our Observations: The ensemble’s overall performance appears comparable to the state-of-the-art ExtractFix [35] repair tool which repaired 10 / 12. However, while many provided fixes do appear to fix the program (e.g., see the patches for EF07 in Fig. 13), other ‘repaired’ programs have patches that are implausible (e.g. the patches for EF15, provided in Fig. 17 in the Appendix) – while they ‘fix’ the bugs and the programs pass their regression tests, they...
introduce other bugs which are not adequately tested for. As it is not scalable to closely examine all ‘successful’ programs, we manually examine the ‘highest-confidence’ (as each LLM gives a relative ‘confidence score’ with each output) patches that pass both functional and security testing in Table VII. From this, we hypothesize that a majority of the ‘successful’ programs may be unreasonable. Noting that (i) the LLMs were not trained specifically for repair (i.e., this is a zero-shot setting) and (ii) they can use only the limited context provided in their prompts, their performance is still remarkable. They even manage to convincingly fix one scenario (EF20) which ExtractFix could not.

As before, the performance across LLMs and prompts varies; the OpenAI models outperform the others. Our ‘gpt2-csrc’ is the ‘underdog’; it has far fewer parameters and is trained on a much smaller corpus (see Section II-D). Because we had access to the training data for this model, we checked whether any of the correct fixes it generated were included in the training set, and found that the fix for EF01 was indeed present. We hypothesize that the black-box LLMs might also benefit from this effect.

The projects where LLMs failed have similar characteristics

![Table VII](image)

| Scenario, Engine                  | Pass? |
|----------------------------------|-------|
| code-cushman-001                 | R.    |
| code-davinci-001                 | R.    |
| code-davinci-002                 | R.    |
| j1-large                         | R.    |
| gpt2-csrc                        | R.    |
| polycoder                        | R.    |

*EF09 and *EF15 pass functional tests but are unreasonable patches.

Fig. 12. Results when using Black-box LLMs to patch real-world bugs from ExtractFix [35]. 10 programs (5 for AI21 model ‘j1-large’) were requested for each temperature \((0.00, 0.25, 0.50, 0.75, 1.00) \times \text{top-k} [1.00]\), giving 50 (25 for AI21) possible total programs. The results are presented as ‘safe and functional’/valid (compiling) programs’. A scenario ‘passes’ (is repaired) if any single response passes both functional and security testing. Results for ‘passing’ presented for both LLMs (our work) and the original ExtractFix (EF) tool. *EF08 and *EF15 pass functional tests but are unreasonable patches. See Table VII.

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
We caution that many of the ‘successful’ results noted in our study on the question “How reliable are LLMs at generating repairs?” we find the answer to be mixed. In general, we are surprised by the quality and success of the code generation by the LLMs when tasked with security bug repair, considering that 100% of the synthetic and hand-crafted scenarios were convincingly repaired. However, based on the qualitative analysis of the real-world scenarios, we do not yet think the LLMs are sufficiently reliable to usurp automatic program repair. Further, other LLM-based constraints apply: repairs are restricted to a single place within a single file, and although security bugs tend to be more localized than other bugs [17], this is not universal.

VII. STUDY LIMITATIONS

Potentially Inadequate Functional Tests. A well-known problem in program repair is that regression tests for a project are weak proxies for the correctness of the program. For example, Qi et al. [42] found that although GenProg [43] claimed to fix 55 bugs, with stricter testing only two of the patches were correct. As noted in Table VII, this limitation applies to patches generated by evaluated LLMs. If testing suites were not comprehensive, programs that appear ‘repaired’ may not truly be so. Future work should consider robust approaches to evaluation (e.g., fuzzing).

Security Tool Test Dependency. Evaluating the LLM patches also required pairing them with bug detection tools. For the initial experiments (synthetic and hand-crafted scenarios) we used CodeQL, which has also been used by others [12]. To evaluate its performance, we performed extensive manual validation during the set-up phases of this project, and every bug that was repaired was built from bug reports generated by CodeQL itself (i.e., in Fig. 2 both ‘Security Tests’ are implemented by CodeQL running the exact same queries). To minimize author-introduced bugs in the tooling, we relied on CodeQL’s repository of existing queries. Of our software-based synthetic and hand-crafted scenarios, all could be analyzed by CodeQL.

For the real-world scenarios we rely on crashing inputs (in ASAN/UBSAN). We ensured that the input that caused the failing program behavior as reported by ASAN was tested (a) against the original program (to observe failing behavior), and (b) against the “repaired” program (to see if it still fails). A pass guarantees that the failure case is repaired, however does not guarantee that the vulnerability is repaired, as it is impossible for testing to prove a bug’s absence.
Scenario Design. While we tried to capture a wide variety of different security weaknesses in different scenarios, there remain many languages and weaknesses not considered.

Prompt Engineering. While we designed several prompt templates, other approaches can be explored. We did not explore cases whereby multiple files needed to be patched (or comprehended) simultaneously. Deciding which context to provide given limited LLMs tokens is an open challenge.

Vulnerability Disclosure. We use synthetic, hand-crafted, and historic vulnerabilities, so no disclosure was required.

VIII. RELATED PRIOR WORK

Studies in the design and maintenance of software systems has produced a vast body of research, such as insights into the nature of security bugs/weaknesses [32] and security patches. For example, security patches are more localized and with fewer source code modifications compared to non-security bugs [17]. Other work has examined practices around comments in code [9]–[11]. Comments serve several purposes, from notices through to explanations of parts of the code for other developers [9], [10]. While comments can provide useful context, they can also mislead developers [11]. Researchers studied commented-out code [29], where code is preserved (sometimes temporarily) over the course of debugging, adding functionality, and other stages of software design.

Given increasing complexity of software and the ongoing pursuit of development process efficiency, researchers have investigated myriad ways to support software development, such as tools for code completion [1], [3], [44]–[46] or recommending fixes based on compiler error messages by “crowdsourcing” collective wisdom/experience [47], [48]. Other prior work includes techniques and tools for detecting security bugs [19], [30], [49], [50]. Exploiting our collective experiences for helping software designers is, in a sense, exemplified by emerging machine learning (ML)-based approaches including the LLMs we investigated [1], [8], [25], given their training on large quantities of open-source code. Such models have an enormous number of parameters and are offered as a black-box tool for software developers to try. To date, we are aware of only the study by Pearce et al. [12] that applies a security lens to gauge the security implications of AI-generated code completions in a large, black-box product (GitHub Copilot), although without considering functional correctness.

While this study investigates using LLMs to recommend fixes for security bugs, there is broader research in automatic software repair. For a survey, consider work by Monperrus [22]. Literature in this area deals with the repair of programs that violate a specification (e.g., as captured by a functional test suite), of which repairing security bugs is one specialization [35]. Approaches include those based on symbolic execution and formal properties [35], matching code to a database of bug-fix scenarios [48], and NMT-based approaches for bug fixes (e.g., [13], [14], [51], [52]).

NMT-based approaches train dedicated models, such as recurrent neural network (RNN)-based encoder-decoder architectures [15] on explicit “bug-fix pairs” mined from software repositories (e.g., through finding code changes through version control logs). Tufano et al.’s approach learns patches for individual functions, with “less than 100 tokens” and demonstrated some potential in generating fixes (they reported 9.22% on their test set). Their approach does not consider comments. DeepDebug [16] requires the curating of bug-/patch datasets and formulates the goal of predicting human-designed patches. An approach by Jiang et al. first pre-trains a model on a large software code-base to “learn developer-like source code” before fine-tuning to specialize on the program repair task [14]. They propose techniques to improve the search for viable patches amongst several predicted tokens. SequenceR [13] focuses on “one line fixes” by curating a dataset of commits that modify single lines of code as well as a copy mechanism for working around the model architecture’s limited vocabulary. NMT-based approaches steer clear of problems due to “overfitting” test suites [42], a pitfall of generate-and-validate approaches for patch generation (e.g., the seminal GenProg [43]). By setting the target for repair models to be (re)-production of human-generated patches, the expectation is that human-generated patches are more likely to be correct (leading to correct outputs) rather than plausible (“passes” test suites without guarantees of correct output). Human-generated security patches are not always correct [17].

Generally, prior work has restrictions (e.g., single line fixes or language-specificity). Hence, we investigated larger, “general-purpose” LLMs and tried to characterize their ability in the security bug fixing domain, without specialization. Our approach of probing the models as bug-fix recommenders, without fine-tuning, bears some similarity to studies of LLMs in different natural language processing (NLP) tasks [6].

IX. CONCLUSIONS AND FUTURE WORK

In this paper we set out to characterize large language models (LLMs) for code in terms of their ability to repair software vulnerabilities in a zero-shot setting. We found that general purpose, black-box LLMs can generate fixes for security bugs when given a carefully constructed prompt, including in 100% of our synthetic and hand-crafted scenarios. However, our evaluation of the LLM’s performance shows that the state of the art is not yet enough for the approach to deliver real value in a program repair framework.

While we believe future, targeted LLMs have potential for use in this area, challenges remain. For a full end-to-end system, complete systems will need to incorporate bug localization and improved test rigs for evaluation. Querying an ensemble of LLMs is unlikely to scale for large numbers of developers to adopt in their day-to-day workloads.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive feedback and encouragement. This work was partially supported by ONR Award # N00014-18-1-2058 and NSF award # 2145482. Any opinions, findings, and conclusions, or recommendations expressed are those of the authors and do not necessarily reflect the views of the sponsors.
in Production Systems,” in 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, Aug. 2021, pp. 1965–2006. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/yagmann.

[38] A. Zeller and R. Hildebrandt, “Simplifying and Isolating Failure-Inducing Input,” IEEE Trans. Softw. Eng., vol. 28, no. 2, pp. 183–200, Feb. 2002, publisher: IEEE Press. [Online]. Available: https://doi.org/10.1109/32.988498

[39] J.-D. Choi and A. Zeller, “Isolating Failure-Inducing Thread Schedules,” in Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSSTA ’02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 210–220, event-place: Roma, Italy. [Online]. Available: https://doi.org/10.1145/566172.566211

[40] S. K. Sahoo, J. Criswell, C. Geigle, and V. Adve, “Using Likely Invariants for Automated Software Fault Localization,” in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS ’13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 139–152, event-place: Houston, Texas, USA. [Online]. Available: https://doi.org/10.1145/2451116.2451131

[41] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea, “Failure Sketching: A Technique for Automated Root Cause Diagnosis of in-Production Failures,” in Proceedings of the 25th Symposium on Operating Systems Principles, ser. SOSP ’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 344–360, event-place: Monterey, California. [Online]. Available: https://doi.org/10.1145/2815400.2815412

[42] Z. Qi, F. Long, S. Achour, and M. Rinarid, “An analysis of patch plausibility and correctness for generate-and-validate patch generation systems,” in Proceedings of the 2015 International Symposium on Software Testing and Analysis. Baltimore MD USA: ACM, Jul. 2015, pp. 24–36. [Online]. Available: https://dl.acm.org/doi/10.1145/2771783.2771791

[43] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “GenProg: A Generic Method for Automatic Software Repair,” IEEE Transactions on Software Engineering, vol. 38, no. 1, pp. 54–72, Jan. 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6035728/

[44] J. Austin, A. Odena, M. Nye, M. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton, “Program Synthesis with Large Language Models,” arXiv:2108.07732 [cs], Aug. 2021, arXiv:2108.07732. [Online]. Available: http://arxiv.org/abs/2108.07732

[45] R. Mihalcea, H. Liu, and H. Lieberman, “NLP (Natural Language Processing) for NLP (Natural Language Programming),” in Computational Linguistics and Intelligent Text Processing, A. Gelbukh, Ed. Springer Berlin Heidelberg, 2006, pp. 319–330.

[46] R. Drachsler, I. G. Harris, and R. Wille, “Generating formal system models from natural language descriptions,” in IEEE Int. High Level Design Validation and Test Workshop (HDLVT), 2012, pp. 164–165.

[47] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, “What would other programmers do: suggesting solutions to error messages,” in Proceedings of the 28th international conference on Human factors in computing systems - CHI ’10. Atlanta, Georgia, USA: ACM Press, 2010, pp. 1019. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1753326.1753478

[48] 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/

[49] A. Barera, T. Rathavarsara, N. Weerawarana, C. Elivioglu, K. Thilakaratna, 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.

[50] 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

[51] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “An empirical investigation into learning bug-fixing patches in the wild via neural machine translation,” in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, Sep. 2018, pp. 832–837. [Online]. Available: https://doi.org/10.1145/3238147.3240732

[52] T. Luellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, “CoCoNuT: combining context-aware neural translation models using ensemble for program repair,” in Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, Jul. 2020, pp. 101–114. [Online]. Available: https://doi.org/10.1145/3395363.3397369

APPENDIX

Source and Dataset Access
We provide the data and code used for this manuscript at the Zenodo link: https://zenodo.org/record/7199939 [31].

Supplementary Figures and Tables

| Scenario | # Gen. | # Vld. | # Fn. | # Vuln. | # Fn. & Vuln. | # Safe. |
|----------|--------|--------|------|-------|-------------|-------|
| CWE-787  | 500    | 440    | 410  | 452   | 388         | 22    |
| CWE-89   | 500    | 491    | 23   | 23    | 468         |       |

Gen. (Generated), Vld. (compilable), Vuln. (Vulnerable), Fn. (Functional), Safe (Not Vulnerable)

TABLE VIII SYNTETIC VULNERABLE PROGRAM GENERATION RESULTS

| Scenario, Engine | Prompt Template |
|------------------|----------------|
| code-cushman-001 | n.h. 137/138 |
| code-davinci-001 | s.1 127/128 |
| code-davinci-002 | s.2 125/126 |
| code-davinci-002 | c. 211/213 |
| code-davinci-002 | c.n. 223/225 || (a) Sums of hand-crafted repair totals from Fig. 6.

Prompt Template

| Scenario, Engine | Prompt Template |
|------------------|----------------|
| code-cushman-001 | n.h. 10/11 |
| code-davinci-001 | s.1 17/18 |
| code-davinci-001 | s.2 16/17 |
| code-davinci-001 | c. 14/15 |
| code-davinci-001 | c.m. 26/27 |
| j1-large         | 5/113 |
| j1-jumbo         | 10/164 |
| gpt2-csrc        | 8/108 |
| polycoder        | 0/12 |

(b) Sums of real-world repair totals from Fig. 12.

Fig. 14. Sums of ‘functional and safe’/valid (compiling) programs’ for the hand-crafted and real-world repair scenarios for each template.

TABLE IX REAL-WORLD PROGRAM REPAIR: TEMPLATE PERFORMANCE. HIGHER VALID REPAIR % (I.E. # FN & SAFE / # VLD) ARE BETTER

| Template ID | # Gen. | # Vld. | # Vuln. | # Fn. & Vuln. | # Safe. | % Vld Repair |
|-------------|--------|--------|--------|-------------|-------|-------------|
| s.1         | 3250   | 868    | 452    | 582         | 446   | 136         |
| s.2         | 3250   | 736    | 521    | 521         | 345   | 176         |
| c.          | 3250   | 743    | 373    | 523         | 363   | 160         |
| c.a.        | 3250   | 1308   | 854    | 935         | 769   | 166         |
| c.n.        | 3250   | 1323   | 937    | 1017        | 871   | 146         |

Gen. (Generated), Vld. (compilable), Vuln. (Vulnerable), Fn. (Functional), Safe (Not Vulnerable)
Fig. 15. CWE-1271 (Verilog with uninitialized reset value) program repair. Highlighted code was generated by LLMs.
Fig. 17. Highest confidence ‘successful’ repair patches for EF15.

Fig. 18. Highest confidence ‘successful’ repair patches for EF01.
Fig. 19. Highest confidence ‘successful’ repair patches for EF20.