USENIX'23 Artifact Appendix: Lost at C: A User Study on the Security Implications of Large Language Model Code Assistants

Gustavo Sandoval[*], Hammond Pearce[*], Teo Nys, Ramesh Karri, Siddharth Garg, Brendan Dolan-Gavitt
New York University

A Artifact Appendix

A.1 Abstract

This artifact contains raw user data collected during the user study and the scripts used for study evaluation. The user data includes (1) anonymous demographic information, (2) submitted code files, (3) the annotations to those code files performed by the authors' manual security analysis, (4) the complete database record of 'prompts' and 'suggestions' by the utilized language model (`code-cushman-001` by OpenAI). The scripts for study evaluation are written in Python version 3.10.6, and may be executed on any compatible machine.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The code executes locally on resources obtained via the artifact repository. No internet or network access is used beyond this initial download of the artifact repository and the subsequent installation of software dependencies (Python libraries, see Section A.2.4).

The user study data was collected via the involvement of human participants and was approved by New York University's Institution Review Board (IRB) as #IRB-FY2022-6074. There is no known risk for evaluators when executing this artifact as no destructive steps are taken and no evaluator files will be impacted.

A.2.2 How to access

Access via URL: https://zenodo.org/record/7187358

A.2.3 Hardware dependencies

No special hardware dependencies are required, only a machine capable of running Python. The authors used a computer with 16GB of RAM and an Intel i7-10750 with Ubuntu 22.04.

A.2.4 Software dependencies

We assume that evaluation is being undertaken on a Debian-based Linux system. The specific software dependencies are Python 3.10.6, virtualenv 20.13.0, and pip 20.0.2. Installation is described under Section A.3.1.

[*]Equal Contribution

A.2.5 Benchmarks

The user study data is provided as an input to the artifacts. They are provided in the folder `data` as part of the repository download.

A.3 Set-up

A.3.1 Installation

Ensure build-essential, Python3, pip, virtualenv, and parallel are installed. These should be able to be installed on Debian-based Linux systems with:

```
$ sudo apt-get install build-essential
$ sudo apt-get install python3 python3-pip parallel
$ sudo pip3 install virtualenv
```

Download the repository from the Zenodo URL. Navigate to the root of the downloaded folder, and create and activate a new virtual environment:

```
$ virtualenv venv
$ source venv/bin/activate
```

Now install the necessary Python libraries:

```
$ pip install -r requirements.txt
```

A.3.2 Basic Test

You can test that your system works by running the first (and simplest) generation,

```
$ python plot_fig7.py
```

This should produce:

```
Created FIG7 as figures/functionality.pdf
```

Which you can check by opening that file and seeing that it matches the paper.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1 / RQ1 - Functionality): There are systematic differences between the 'assisted' and the 'control' groups, with the 'assisted' group having a small but consistent advantage over the 'control' group, and the 'autopilot' group outperforming both. The small sample size however means that the comparison does not reach statistical significance. These results are presented in the paper in Section 4.2.Results and Figure 7, and reproduction is described in Experiment E1 of this Appendix.

(C2 / RQ2 - Security 'aggregate'): For all four cases {CWEs/LoC over compiling functions; CWEs/LoC over functions passing unit tests; Severe CWEs/LoC over compiling functions; Severe CWEs/LoC over functions that pass the unit test} the 'assisted' group has fewer bugs compared to the 'control', with up to a 22% lower mean for the 'assisted' group compared to the 'control' for Severe CWEs/LoC over functions that pass unit tests. For severe CWEs, the comparisons are also statistically significant using non-inferiority tests with $\delta = 10\%$. This suggests that in the aggregate case, LLMs may provide a slight benefit to security. These results are presented in the paper in Section 4.3.3 Topline results and Figure 8, and reproduction is described in Experiment E2 of this Appendix.

(C3 / RQ2 - Security 'per-function'): When examining individual functions in the user study, results vary, with different functions being harder or easier for each group and with some differences being statistically significant. As a result of this analysis, it is hard to conclude how LLM suggestions may impact the code security of any arbitrary code-writing task (some functions are made less buggy, some are made more, it likely depends on the complexity of each specific function). These results are presented in the paper in Section 4.3.4 Per-function CWE rates, and Table 3, and reproduction is Experiment E3 of this Appendix.

(C3 / RQ3 - Bug origin): Even though suggestions from the LLM may contain bugs, the human developers introduced a majority of the bugs present in the submitted code from the 'assisted' group. These results are presented in the paper in Section 4.4 RQ3 - On the origin of bugs.

A.4.2 Experiments

The functionality of these artifact scripts are predicated on the completion of a user study as outlined in Preliminary 1 and the formatting of that data in Preliminary 2 as well as bug data encoding in Preliminary 3. Should a fresh user study not be desirable/attempted, evaluators may skip ahead to the data analysis stage starting with Experiment E1.

Preliminaries (User study and first-pass processing):

Preliminary 1 - User study (Manual) [Approximately 1 month]: Each experiment E1-E4 implicitly relies on the data produced by a user study following the setup of the paper (see the paper Section 3). This would use the user study participant files provided in the linked artifacts repository folder `study_participant_instructions`. The data produced by our user study is provided.

Preliminary 2 - First-pass data processing (Automated) [Approximately 15 compute-minutes]: This step converts the given user study data files into ones suitable for the rest of the data processing pipeline. This includes running functional tests against the study-designed test cases and breaking the files into separate functions for split testing (see paper Section 4.2 'Split Testing') and manual bug encoding (see Preliminary 3). Note that executing the linked scripts 'resets' the bug encoding process: as a result, the script will create a new directory rather than overwrite the data that our study collected (should you wish, you may manually overwrite our data by using a copy paste).
Preparation: As per Section A.3.1.
Execution: From the `functionality_tests` directory, execute `./run_all.sh`. This script creates a directory with the necessary testing infrastructure for each user study file in `functionality_tests/repos/{uuid}`, executes the test suites, and saves the results to JSON files in each directory named `api_report.json` (indicating which API functions were implemented, unimplemented, or failed to compile), `orig_testsuite.json` (the results of the basic 11-function test suite), and `ref_testsuite.json` (the results of the expanded 45-function test suite).
This script will produce for each participant a directory containing the results of the split-functional testing and the components of each of their submissions. It also creates extraneous/intermediate files we removed from our own data output folder, which can be found at `data/submitted_assignments/*`.

Preliminary 3 - Bug encoding (Manual) [Approximately 66 person-hours]: To evaluate the security of the code provided by the users, manual analysis was performed. This involved the process described in the paper Section 4.3.1. From a technical point of view, each file in the `data/submitted_assignments/{uuid}/parts/gen_{function}.c` was read by three people who manually looked for security relevant bugs. These annotations are still present and may be modified or evaluated by artifact evaluators. Once this evaluation is done, the bugs must be 'typed up' into a sqlite database with the schema described in `bugs_and_demographics.sqlite3`.

Analysis of user study data:

(E1 / RQ1 - Functionality) [<1 minute total]: This experiment runs several Python scripts.

Preparation: Section A.3.1 and the preliminaries.

Execution: To see the data relevant to this claim run the command: `python plot_fig7.py`. This will produce the file `figures/functionality.pdf` as well as the file `data/derived_data/functionality_stats.txt`.

Results: The file `figures/functionality.pdf` displays the relationship between groups, and should show how the 'autopilot' group appears to function better than the 'assisted' group, which functions better then the 'control' group. Statistically speaking, however, the comparisons between groups for code passing basic and expanded tests does not reach statistical significance as shown in the `functionality_stats.txt` file from line 88 onwards.

(E2 / RQ2 - Security 'aggregate') [<1 minute total]: This experiment runs several Python scripts.

Preparation: Section A.3.1 and the preliminaries.

Execution: To see the data relevant to this claim run the command: `python plot_fig8.py`

Results: The following results will be presented in the terminal and they will create the appropriate images for Figure 8 of the paper:

```
Plotting figures/bugs_per_loc_compiled.pdf
Plotting figures/bugs_per_loc_passing.pdf
Plotting figures/bugs_per_loc_severe_compiled.pdf
Plotting figures/bugs_per_loc_severe_passing.pdf
```

In addition, to run the non-inferiority tests with $\delta = 10\%$, after running the script to run the figure, you may run the script `python inferiority_tests.py`. This script will produce in the command line the results that show the *p-values* for each of the four groups. The two important values for the graph are the values for `Per Loc Severe Compiled` and `Per Loc Severe Passing`, which show that for 'the Severe CWEs per Line of Code Compiled' the non-inferiority test is significant with *p-value* of p=0.04 and the Non-inferiority test for the Severe CWEs per LOC passing which is p=0.06.

(E3 / RQ2 - Security 'per-function') [<1 minute total]: This experiment runs several Python scripts.

Preparation: Section A.3.1 and the preliminaries.

Execution: The results for this claim are made in a two step-process. Firstly, we derive the main Table 3 data by the command `python generate_table3.py` which will produce the file `data/derived_data/table3.tsv`. Then, we perform the non-inferiority tests by running `python inferiority_per_func.py`.

Results: The statistical test results between function group pairings are directly printed to the console. These in conjunction with the derived `table3.tsv` should match Table 3 in the paper.

(E4 / RQ3 - Bug origin) [<1 compute-minutes, up to 6 person-hours]: This experiment runs several Python scripts and may involve a further optional human-annotation step.

Preparation: Section A.3.1 and the preliminaries.

Execution: Demonstrating this is a two-step process. First we run the command `python suggestion_cover.py -o data/derived_data/suggestion_cover.html` to display the origin of code in the final output files. There is now a human-annotation step, which is required as determining the relationship with the final code and the model suggestions was often beyond simple lexical analysis. For result reproduction, this step may be considered optional as it is laborious. For each bug present in each file in `data/submitted_assignments/recombined_list_files/Active/{uuid}-list.c`, one must scroll to the given bug location in the `suggestion_cover` file. Using a mouse-over, this will display the lexical origin of the suggestion. If the reviewer agrees, then this annotation can be recorded in the original file in the manner described in `bug_origin_all.py`. The authors thus went through each identified bug and visually determined their origin.

Secondly, once bugs were annotated, we can then use the command `python bug_origin_all.py`, which uses `grep` to scan the annotated code files for the human-annotated bugs and their origins and aggregates the statistics. The script prints the results to the terminal, reporting that 63.1 % of bugs come from human developers in the assisted groups.

Results: Results are presented in the terminal and display the count of the origin of each bug.

We can explore the specific reasons for this further by examining a specific bug, CWE-416, with the command `python bug_origin_cwe416.py`. This will, for each user in the assisted group, find each instance of the bug—then reporting if the first time it appeared it was in a suggestion or in the human's own-written code, then count the number of times it was suggested, accepted, and the number of times it appeared in the final document. The results show that for this bug, the LLM typically suggested the bug even if it was not already present in the user's code.

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.