USENIX Security '26 Artifact Appendix: BugAuditor: Detecting Bugs via Inconsistent Defensive Code Auditing

Miaoqian Lin
The University of Hong Kong

Kai Chen
Institute of Information Engineering, Chinese Academy of Sciences

Hao Chen
The University of Hong Kong

A Artifact Appendix

A.1 Abstract

BugAuditor is an LLM-driven bug detection framework that uses inconsistent defensive handling as a new oracle for detecting project-specific bugs. Its key insight is that large software systems already contain abundant defensive code, where developers apply defensive operations to prevent bugs in security-sensitive contexts. When similar security-sensitive behaviors are handled defensively in some places but not in others, the inconsistency may indicate a real bug.

BugAuditor first identifies defensive code snippets across the codebase, then infers defensive patterns that capture both the security-sensitive behavior and the required defensive handling. It finally applies these patterns to audit similar code contexts and detect inconsistent defensive handling.

This artifact includes the implementation of BugAuditor, key experimental data , and usage documentation. It aims to demonstrate that BugAuditor can effectively locate defensive code in large-scale systems, mine defensive patterns, and detect new bugs.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact poses no risks to the evaluators' machine in terms of security, data privacy, or any other ethical concerns

A.2.2 How to access

You can access the artifacts via the following stable link to Zenodo: https://doi.org/10.5281/zenodo.20267685 You can also access it in Github: https://github.com/Yuuoniy/BugAuditor The artifact is licensed under Apache License 2.0.

A.2.3 Hardware dependencies

At least 4 CPU cores recommended (20+ better). At least 16 GB RAM recommended. At least 15 GB free disk space recommended. BugAuditor works on standard machines. It is implemented in a multithreaded manner, and we recommend using a multithreaded computer to speed up the evaluation process. In our evaluation, we conducted experiments on a 64-bit Ubuntu server equipped with 212 GB of memory, 2 TB of storage, and an Intel Xeon Gold 5218 CPU with 64 cores.

A.2.4 Software dependencies

BugAuditor is implemented in Python and leverages existing code analysis tools, including Joern, Tree-sitter, and Weggli. We provide a Dockerfile to automate the setup and creation of the Docker environment. This Dockerfile includes everything needed to configure the runtime environment, install third-party tools and dependencies, set up Python libraries.

The artifact requires access to LLM APIs. The overall token cost is very low and is convenient for evaluation. The paper uses DeepSeek-V3.2 deployed locally. Due to the official DeepSeek API no longer providing access to DeepSeek-V3.2, we recommended DeepSeek-V4-Flash (via api.deepseek.com) as a convenient replacement. This model produces similar results while maintaining equal or lower cost. DeepSeek-V3.2 can still be used for testing via other available OpenAI-compatible endpoints.

A.2.5 Benchmarks

Source code of the Linux kernel.

A.3 Set-up

A.3.1 Installation

We provide a `Dockerfile` to simplify the installation process. Users can build the BugAuditor Docker image using the provided `Dockerfile` and download the necessary programs for testing. Detailed instructions are available in the `INSTALL.md` file. Please refer to it and follow the steps accordingly.

A.3.2 Basic Test

We provide a minimal running example for a quick sanity check of BugAuditor. This example uses `clk_put` as the seed defensive operation, reproducing the working flow of

defensive code locating, defensive pattern reasoning, and bug auditing on a small input.

1. Basic test for defensive pattern reasoning. [2 human-minutes + 5 compute- minutes]: Run the following command:

```
$ bash artifact/minimal/run_pattern_reasoning.sh
```

The script locates defensive-code snippets for `clk_put`, selects 10 examples, and runs defensive pattern reasoning. The expected terminal output is provided in the file `expected_pattern_reasoning_output.txt` located at `artifact/minimal/reference/`. The main generated results are stored under `artifact/results/minimal/`:

- `defensive_code_snippets.json`
- `inferred_defensive_patterns.json`

The inferred patterns cover multiple security-sensitive operations, including `of_clk_get_by_name`, `clk_hw_get_clk`, `of_clk_get`, `clk_get`, `of_clk_get_from_provider`, and `clk_get_sys`. Note that the minimal example is not meant to exactly reproduce Table 9 in the paper.

2. Basic test for bug auditing. [2 human-minutes + 5 compute- minutes]: Run the following command:

```
$ bash artifact/minimal/run_bug_auditing.sh
```

This test uses one mined defensive pattern, `of_clk_get_by_name -> clk_put`, to audit a focused set of `clk_put`-related cases. It detects buggy functions such as `lpc32xx_clk_init` and `nmdk_timer_of_init`. The key results are written to the `artifact/results/minimal/` directory, including the generated bug report file `bug_reports_clk_put.json`.

A.4 Evaluation workflow

A.4.1 Major Claims

The main workflow uses a given defensive operation as a seed to locate similar defensive code snippets across the codebase (e.g., Linux kernel). It then reasons about the defensive patterns and applies them to detect new bugs. Overall, the process consists of two phases: defensive code locating and pattern reasoning, followed by bug detection.

(C1): BugAuditor can locate a large number of defensive code snippets and infer the corresponding defensive patterns, using six given defensive operations as seeds. This is supported by the experiments (E1).

(C2): Using the inferred defensive patterns, BugAuditor detects dozens of bugs in the Linux kernel. This is validated by the experiments (E2).

A.4.2 Experiments

The experiments were conducted on a reduced scale for faster and more convenient evaluation. Complete experiment scripts and detailed instructions are also provided in `AE.md`.

(E1): [Defensive Pattern Reasoning] [5 human-minutes + 1 compute-hour]:
This experiment reproduces the defensive pattern reasoning stage on the six seed defensive operations used in the paper. It involves defensive code collection and the corresponding pattern reasoning.
Preparation: Follow the previously instructions for installation and downloads the source code of Linux kernel. Run the basic test to check the environment. Make sure setup the LLM endpoint.
Execution: Please run:
`$./artifact/r_pattern_reasoning/run.sh`
Results: The key results are saved under `artifact/results/r_pattern_reasoning/`. `inferred_patterns.csv` lists all defensive patterns inferred, with one pattern per row.

(E2): [Bug Auditing] [5 human-minutes + 30 compute-minutes]:
Preparation: Follow the artifact installation instructions and run the minimal example first to check the environment. The LLM endpoint should be configured before running this stage.
Execution: Please run:
`$ ./artifact/r_bug_auditing/run.sh`
Results: The key results are saved under `artifact/results/r_bug_auditing/`, including the main output file `bug_reports.json`.

A.5 Notes on Reusability

To facilitate artifact evaluation, we designed the reduced-scale evaluations that reduces time and token consumption. We additionally provide instructions for applying BugAuditor to other programs, such as OpenSSL and FFmpeg. Please refer to `AE.md` for details.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.