USENIX Security '26 Artifact Appendix: BenchChecker

Di Wu, Xu He, Shu Wang, and Kun Sun

A Artifact Appendix

A.1 Abstract

BenchChecker is an artifact for assessing data contamination in LLM bug-fixing benchmarks. It implements the paper's two-stage workflow: Stage I probes a target model for repository-presence signals using signature query/target pairs, and Stage II checks patch presence by comparing benchmark patches with model-cutoff repository snapshots, using GumTree-backed localization, embedding similarity, replacement-diff construction, and optional container test-suite verification. The artifact also includes packaged calibration data, benchmark loaders, tests, and evaluator-facing documentation. The final artifact received the Artifacts Available and Artifacts Functional badges.

A.2 Description & Requirements

The final evaluated artifact is BenchChecker-AE v1.1.7. It contains the Python package, command-line interface, benchmark loaders, Stage I and Stage II implementations, unit tests, documentation, and vendored external tools: NiCad 6.2, FreeTXL for Linux-native NiCad runs, and GumTree 4.0.0-beta4. The release also includes a Dockerfile for running BenchChecker without installing its Python and Java dependencies directly on the host. Version 1.1.7 filters invalid empty Stage I query/target pairs before model probing and defensively rejects an empty prompt. This addresses the zero-length Qwen input reported during evaluation. The main AE entry point is:

```
python -m benchchecker.cli full \
  BENCHMARK CASE_ID TARGET_MODEL
```

The command clones the benchmark repository into a workspace, resolves the target model cutoff snapshot, runs both stages, and writes JSON/JSONL outputs under `workspace/runs/`.

A.2.1 Security, privacy, and ethical concerns

The artifact is not destructive and does not intentionally disable security mechanisms. It may clone public repositories, download public HuggingFace models/datasets, install Python packages, and run benchmark test suites in Docker containers when configured. Evaluators should run it in a normal isolated artifact-evaluation workspace and should not place secrets in that workspace except an optional `HF_TOKEN` if their HuggingFace access requires one. Full verification from the supplied container can mount the host Docker socket; this gives the container control over the host Docker daemon. Use the documented restricted mode if that access is not acceptable. No author-owned hardware, reviewer identity, or personally identifying information is required.

A.2.2 How to access

The final stable URL is the permanent Zenodo concept DOI:

`https://doi.org/10.5281/zenodo.20385622`

This DOI represents all versions and resolves to the newest published one. The final file is `BenchChecker-AE-v1.1.7.zip`. Its SHA-256 is:
`73a79296bb935cf349fc98600c4246a58`
`b5145f5774fcf10a66069c25f6aa1d4`

A.2.3 Hardware dependencies

No specialized hardware is required for the functional smoke tests; the `tiny` model profile can run on CPU. For full-scale runs with real target models and faster embedding similarity, we recommend Linux x86_64 with an NVIDIA CUDA GPU and 20–50 GB free disk for repository clones, model caches, and Docker runs. Reviewers can use evaluator-owned compute; no remote author-owned platform is needed.

A.2.4 Software dependencies

Recommended: Ubuntu 22.04 or 24.04 LTS, Python 3.10 or 3.11, Git, Java 17 or newer for GumTree, Docker Engine with a running daemon for container verification, and a C/C++ build toolchain only as required by Python wheels or benchmark dependencies. Windows 11 and macOS are supported for source-level runs, but Docker Desktop is required for containerized benchmark tests and NiCad may use the Docker fallback on non-Linux hosts. Python dependencies are declared in `pyproject.toml`; the optional `artifact` extra installs the SWE-bench harness. The supplied Dockerfile provides a fixed Python 3.11, Java 17, NiCad, FreeTXL, GumTree, and Docker CLI environment.

A.2.5 Benchmarks

The artifact supports network-backed loaders for SWE-bench, SWE-bench Verified, SWE-bench Pro, and Multi-SWE-bench, and path-backed loaders for BugsInPy, BugSwarm, Defects4J,

GrowingBugs, and local JSON/JSONL exports. The archive also includes the original v1.0.0 label and word-frequency assets in `data/`, documented in `data/README.md`.

A.3 Set-up

A.3.1 Installation

Download and unpack the Zenodo archive, then run:

```
python -m venv .venv
. .venv/bin/activate
python -m pip install -e ".[dev,artifact]"
python scripts/install_nicad.py
python -m benchchecker.cli full --help
```

On Windows, use `.venv\Scripts\activate` instead of the POSIX activation command. To inspect Docker/container readiness:

```
python -m benchchecker.cli env doctor
python -m benchchecker.cli env bootstrap \
  --dry-run
```

Alternatively, build and check the supplied CPU container:

```
docker build -t benchchecker-ae:1.1.7 .
docker run --rm benchchecker-ae:1.1.7 --help
```

The restricted and host-Docker modes, GPU option, persistent caches, and mount-path mapping are documented in `docs/CONTAINER.md`.

A.3.2 Basic Test

Run the full workflow with a tiny local model:

```
python -m benchchecker.cli full \
  swebench psf__requests-1724 tiny \
  --workspace workspace/ae-smoke \
  --verification-backend swebench \
  --stage1-max-queries 3
```

Expected outputs include a run directory with a JSON report and generated Stage I pair/probe files. If Docker or the SWE-bench harness is unavailable, the run still records the explicit container status in the report rather than silently hiding the environment issue.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): BenchChecker provides an executable implementation of the two-stage contamination-detection pipeline described in the paper: repository presence followed by patch presence (paper Sections 3–4; E1–E3).

(C2): Across six bug-fixing benchmarks, BenchChecker achieves AUC around 0.9 while keeping the false-positive rate at or below 0.1 (paper Section 5; E4).

(C3): Filtering detected contamination reduces the top-10 systems' reported resolution rates by more than 20% on medium-difficulty tasks in SWE-bench Verified and Multi-SWE-bench (paper Section 6; E4).

A.4.2 Experiments

(E1): Environment and package check, 5–10 human-minutes.

Run `python -m benchchecker.cli full -help`, `python -m benchchecker.cli env doctor`, and optionally `pytest -q`. These commands check that the CLI, package data, vendored tools, and local environment are visible.

(E2): Functional smoke run, 10–30 human-minutes plus compute time.

Run the Basic Test command above. The expected result is a completed run directory with `report.json`, generated Stage I query/probe files, and explicit Stage II verification status. This exercises the main end-to-end code path without requiring a large target model.

(E3): Real-model single case, evaluator-dependent runtime.

For a stronger functionality check, run:

```
python -m benchchecker.cli full \
  swebench django__django-13023 \
  qwen2.5-coder-0.5b \
  --workspace workspace/ae-qwen \
  --verification-backend auto \
  --stage1-max-queries 50
```

The expected result is a populated report for a real model profile. For models not built into `benchchecker/models.py`, provide `-cutoff-date YYYY-MM-DD`. In v1.1.7, generated Stage I pairs have non-empty prompts and targets, so Qwen is not called with a zero-length input.

(E4): Optional reproduction analyses.

Use the reproduction guide, evaluation notebooks, and utility scripts to rerun Stage I, Stage II, baseline, and leaderboard analyses. These runs require the corresponding benchmark data, model access, and substantially more compute than the smoke tests.

A.5 Notes on Reusability

BenchChecker can be used beyond the paper by selecting a supported benchmark loader or by providing a local JSON/JSONL benchmark export with repository, patch, commit, and cutoff metadata. The language-specific preprocessing and configuration files are split under `benchchecker/stage2/preprocess/` and `benchchecker/configs/`, so reviewers can inspect or extend the artifact to additional code-generation benchmarks.

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/.