USENIX Security '26 Artifact Appendix: Exposing Resource-Exhaustion DoS Vulnerabilities with Leak-Oriented Minimum Path Covers

Lige Zhan*
Wuhan University
ligezhan@whu.edu.cn

Yafei He*
Wuhan University
yaf2023@whu.edu.cn

Jiang Ming†
Tulane University
jming@tulane.edu

Guojun Peng*
Wuhan University
guojpeng@whu.edu.cn

Jianming Fu*,‡
Wuhan University
jmfu@whu.edu.cn

A Artifact Appendix

A.1 Abstract

Resource leaks occur when programs fail to properly release scarce GC-unmanaged resources, such as file descriptors, sockets, cursors, and database connections. Repeatedly triggering such leaks may exhaust system resources and cause denial-of-service failures. Prior approaches typically enumerate execution paths to identify resources that are not properly released. However, exceptions may trigger non-local control transfers that bypass resource releases, while exhaustive path enumeration may cause path explosion. To address these limitations, we present LeakHunter, a resource-leak detector for Java programs.

The artifact contains the LeakHunter source code, the DroidLeaks and JLeaks evaluation datasets, a runnable functional validation demo, and a pre-built Docker image.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

LeakHunter performs static analysis of Java source code. It does not execute the analyzed Java programs, exploit vulnerabilities, disable security mechanisms, or perform destructive operations. The public artifact is intended for defensive analysis and does not include automated exploitation or DoS triggers, nor does it provide instructions for attacking third-party systems. Any sensitive details concerning real-world issues are released only after coordinated disclosure and remediation, where applicable.

We recommend using Docker to reproduce the artifact for the Functional evaluation. When the recommended Docker command is used, the container is automatically removed after execution because the command includes the `--rm` option. For a more detailed discussion of ethical considerations, please refer to our paper.

A.2.2 How to access

The artifact is permanently available from Zenodo at:

https://doi.org/10.5281/zenodo.18476387

Please download the latest version of `LeakHunter_upload.zip` from the Zenodo record. To prevent accidental downloads of outdated versions, we have restricted access to the older files and left only the latest version publicly accessible. The `LeakHunter_upload.zip` archive contains:

- `Docker/leakhunter-ae-docker.tar`: pre-built AMD64 Docker image for immediate evaluation;
- `source/LeakHunter-source.zip`: source code, the full DroidLeaks and JLeaks evaluation datasets, Dockerfile, dependency descriptions, and documentation;

A.2.3 Hardware dependencies

We recommend an evaluation environment that meets at least the following hardware requirements:

- an x86-64/AMD64 processor;
- at least 8 GB of RAM, with 16 GB recommended;
- at least 30 GB of available disk space.

No GPU, FPGA, specialized processor, hardware performance counter, or other special hardware is required.

A.2.4 Software dependencies

For the recommended evaluation procedure, the evaluator only needs Docker Desktop or Docker Engine. The supplied Docker image already contains:

- Python 3.9;
- OpenJDK 21;
- Graphviz;
- Python packages required by LeakHunter.

For native source execution, the evaluator must install Python 3.9, OpenJDK 21, Graphviz, the packages listed in `requirements.txt`, and `openpyxl`.

A.2.5 Benchmarks

The complete artifact includes the DroidLeaks and JLeaks datasets used in the paper's evaluation. For the Functional badge evaluation, we provide a small subset, `DroidLeaks-refined_demo.csv`, for quick validation. This subset contains five bug-fix pairs, comprising ten analysis inputs. In each pair, the buggy version contains a resource leak, whereas the corresponding fixed version does not. The subset is intended to validate LeakHunter's main detection pipeline and is not intended to reproduce the complete quantitative evaluation reported in the paper.

A.3 Set-up

A.3.1 Installation

We recommend using the pre-built Docker image. After downloading and extracting `LeakHunter_upload.zip`, enter the directory containing `leakhunter-ae-docker.tar` and load the image with:

```
docker load -i leakhunter-ae-docker.tar
```

A successful import should produce output similar to:

```
Loaded image: leakhunter-ae:latest
```

The evaluator may confirm that the image is available by running:

```
docker images leakhunter-ae
```

Alternatively, LeakHunter can be run natively from the source package. From the extracted source directory, install the Python dependencies with:

```
python -m pip install -r requirements.txt
python -m pip install openpyxl
```

A.3.2 Basic Test

We provide a basic functionality test located at `src/scan_basic_test.py`. It can be executed directly using the following command:

```
docker run --rm leakhunter-ae python -u \
src/scan_basic_test.py
```

If the messages `Detection result: LEAK` and `Basic functionality test passed` appear, LeakHunter has been successfully set up and its main detection pipeline is functioning correctly, as shown in Figure 1.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): After inferring resource-management APIs and resource-relevant exceptions, LeakHunter can detect resource leaks. This capability is demonstrated by its successful detection of the leaks in all five buggy methods included in the runnable demo. This claim is validated by experiment (E1), for which the expected results are `TP=5` and `FN=0`.

(C2): After inferring resource-management APIs and resource-relevant exceptions, LeakHunter can correctly recognize non-leaking methods. This capability is demonstrated by its correct classification of all five fixed methods included in the runnable demo. This claim is validated by experiment (E1), for which the expected results are `TN=5` and `FP=0`.

A.4.2 Experiments

(E1): *Runnable Demo Functional Validation* [approximately 2 human-minutes + 10 compute-minutes + 30 GB Docker disk usage].
This experiment validates claims (C1)–(C2) using five bug-fix pairs, comprising ten analysis inputs.
Preparation: Load the supplied Docker image as described in the Installation section:
`docker load -i leakhunter-ae-docker.tar`
Confirm that `leakhunter-ae:latest` is listed by:
`docker images leakhunter-ae`
Confirm that LeakHunter is functioning correctly, as described in the Basic Test section, by running:

```
docker run --rm leakhunter-ae python -u \
src/scan_basic_test.py
```

Execution: Run the functional validation demo:

```
docker run --rm leakhunter-ae:latest python -u \
  runnableDEMO/scan_droidleaks_demo.py
```

Results: For claims (C1) and (C2), inspect the classification of the buggy and fixed versions. Each buggy version should be reported as `LEAK` and counted as a true positive. Each fixed version should be reported as `NO LEAK` and counted as a true negative.
The expected final statistics are:

```
Metrics so far: TP=5, FP=0, FN=0, TN=5,
       Precision=1.0000, Recall=1.0000
```

Figure 2 illustrates one complete output of the demo.

A.5 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/.

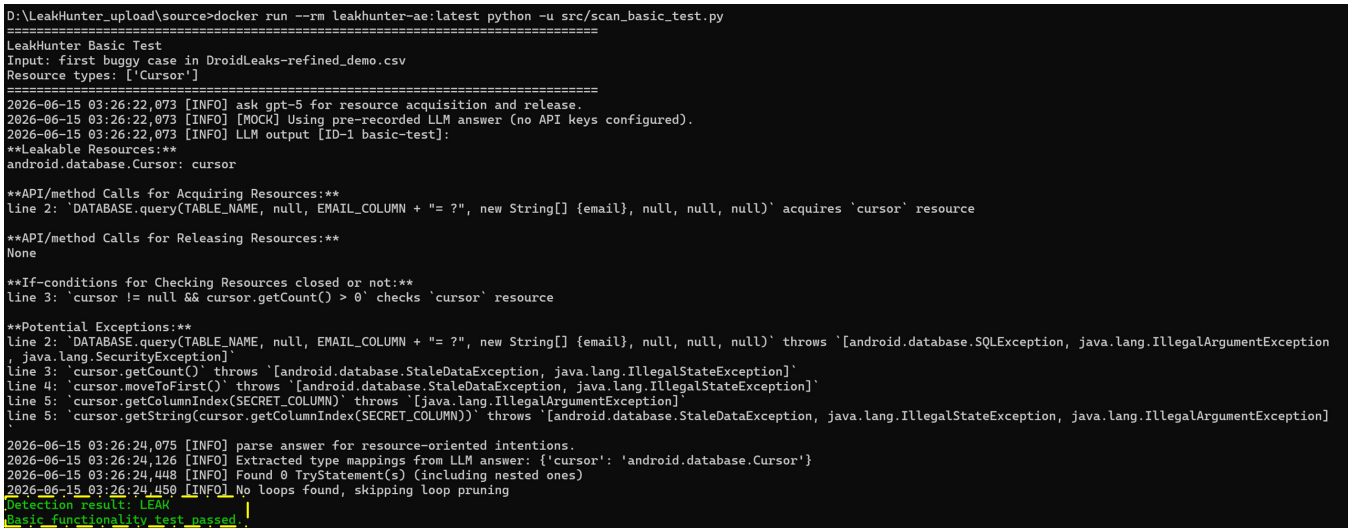

Figure 1: Expected output of the basic functionality test

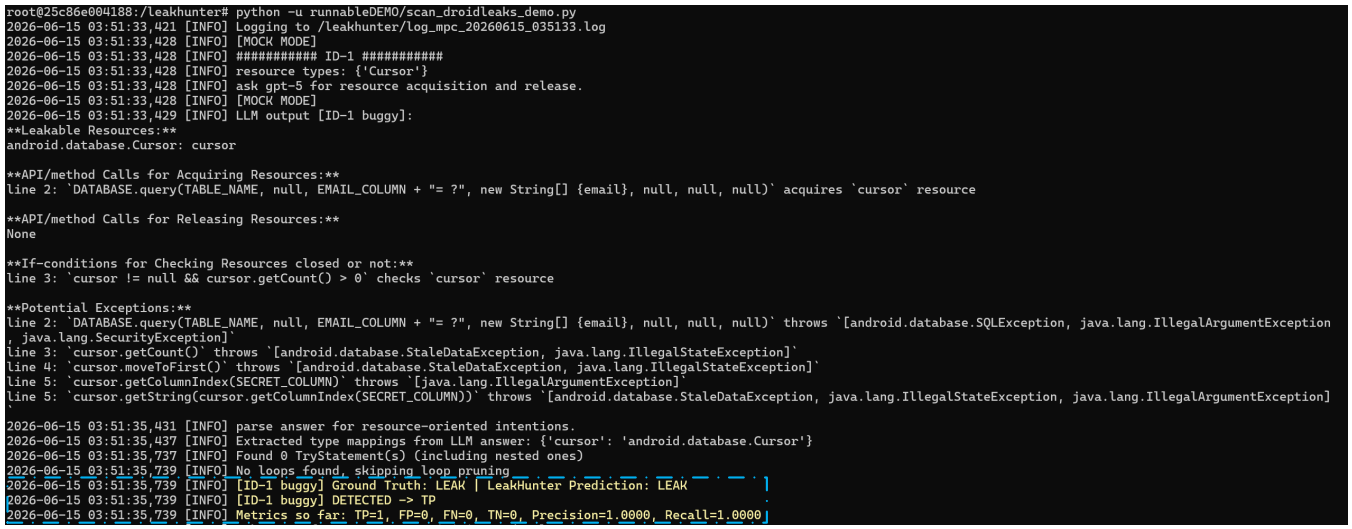

Figure 2: Representative output of the runnable functional validation demo. The output includes the case identifier, identified resource information, leak classification, and cumulative statistics