USENIX Security ’25 Artifact Appendix: My ZIP isn’t your ZIP: Identifying and Exploiting Semantic Gaps Between ZIP Parsers

Yufan You[1], Jianjun Chen[1,2,✉], Qi Wang[1], and Haixin Duan[1,2]

[1]Tsinghua University
[2]Zhongguancun Laboratory

A Artifact Appendix

A.1 Abstract

Our artifact consists of three major components, along with several utility scripts to reproduce the results in the paper:

1. The ZIPDIFF differential fuzzer designed to discover inconsistencies between multiple ZIP parsers. It has two options to reproduce the ablation study in the paper.

2. Docker images for the 50 tested ZIP parsers. We provide both prebuilt image files and source files to build them.

3. Construction code for ambiguous ZIP files. Based on our findings, we classify the ZIP parsing ambiguities into 14 types and constructed sample ZIP files for each variant of them. We use the construction code to demonstrate how to implement the ambiguities described in the paper. We also use them to measure the types of inconsistencies between each of the 1225 pairs of parsers.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All ZIP files in this artifact are only used to test inconsistencies between ZIP parsers. They do not contain harmful payloads such as malware, and the parsers are isolated in Docker containers, so it is safe to run the experiments.

A.2.2 How to access

Our artifact is available on Zenodo[1] and GitHub[2]. On Zenodo, the source code is provided in `ZipDiff.tar.gz`, and the prebuilt Docker images are provided in the `*.tar.bz2` files.

[✉]Corresponding author: jianjun@tsinghua.edu.cn
[1]`https://doi.org/10.5281/zenodo.15526863`
[2]`https://github.com/ouuan/ZipDiff/tree/zenodo-v7`

A.2.3 Hardware dependencies

We recommend running the experiment on a machine with at least 128GB of RAM and 300GB of disk space. More CPU cores will speed up the experiment. The authors used Intel Xeon Gold 6330 with 112 logical cores and 944GB of RAM.

The memory requirement comes from two sources: First, it runs 50 parsers in parallel that may consume around 60GB of RAM at peak. Second, the unzip outputs can sometimes be very large, and the file I/O operations will be slow if it exceeds the cache size. The memory usage also depends on the number of CPU cores, as the parsers will run in parallel based on the number of available CPUs.

The disk space requirement is also mainly caused by large unzip outputs. The outputs are compressed after each iteration, so the final results will be smaller than the peak usage during a single iteration. The disk space requirement will be much lower if the file system supports transparent compression.

The fuzzer provides a batch size option that can be lowered to reduce the resource requirements. However, the Docker overhead will be significant if the batch size is too small.

A.2.4 Software dependencies

The experiments require a Linux system with Rust, Docker and Docker Compose. Python 3 with `numpy` and `matplotlib` is required for generating the tables and figures. The `zip` command will be helpful in the basic test.

The authors used Ubuntu 23.10 with Linux 6.5.0-44, Rust 1.86.0, Docker 27.1.1 with Docker Compose 2.33.1, and Python 3.13.3 with numpy 2.3.0 and matplotlib 3.10.3.

A.2.5 Benchmarks

The tested ZIP parsers are provided as Docker images. The prebuilt image files are available on Zenodo.

A.3 Set-up

A.3.1 Installation

Follow these steps to set up the dependencies and the artifact:

1. **Install Rust.** Follow the instructions at `https://www.rust-lang.org/tools/install` to install Rust using rustup. Either install the latest stable version (by default) or 1.86.0 used by the authors.

2. **Install Docker and Docker Compose.** Follow the instructions at `https://docs.docker.com/engine/install` and `https://docs.docker.com/compose/install/linux`.

3. **Fetch source code.** Either download `ZipDiff.tar.gz` from Zenodo and extract the archive by `tar xf ZipDiff.tar.gz`, or clone it from GitHub by `git clone https://github.com/ouuan/ZipDiff`.

4. **Compile Rust code.** Run `cargo build --release` in the `zip-diff` directory of the source code.

5. **Build or load Docker images.** Build Docker images of the ZIP parsers by running `tools/prepare.sh`, `cd parsers`, and `sudo docker compose build`. Alternatively, load the prebuilt image files: `for i in *.tar.bz2; do docker load -i "$i"; done`. Most parser versions are fixed in the source files, but a few may change, so it is recommended to load prebuilt images to ensure using the same versions as the paper.

A.3.2 Basic Test

Test parsers. Create a ZIP: `zip -0 a.zip README.md`. Run parsers on it: `tools/run-parsers.sh a.zip`. The parsers will print some logs and exit. The outputs of the parsers will be saved in subdirectories of `evaluation/results/a.zip`. All parsers should extract `README.md` successfully.

Minimal fuzzing. Run `sudo target/release/fuzz -b 10 -s 120` in the `zip-diff` directory. This will run fuzzing for two minutes with only ten samples per batch. The fuzzer will print logs for each iteration. The log text should contain `ok: 50`, indicating that all parsers are working fine. The results will be available at `evaluation/stats.json`, `evaluation/samples` and `evaluation/results`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): ZIPDIFF can discover inconsistencies between the majority of the 50 tested ZIP parsers. The chosen system design outperforms the two other setups in the ablation study. This is proven by the experiment (E1) described in Section 5.1 and 5.3, with results plotted in Fig. 4.

(C2): We identified 14 types of ZIP parsing ambiguities as described in Section 5.2. We constructed sample files corresponding to these ambiguities. All 14 types can cause inconsistencies between some pairs of parsers, and 1221 out of a total of 1225 pairs of parsers are affected by at least one type of ambiguity. This is proven by the experiment (E2) and the number of inconsistency types between each pair of parsers is recorded in Table 4.

A.4.2 Experiments

(E1): Fuzzing and ablation study [5 human-minutes + 15 computer-days + 100GB final / 500GB runtime disk]
Preparation: We default to run five 24-hour fuzzing sessions for each of the three setups with a batch size of 500. For quick verification, the duration of each session (`STOP_SECONDS` in `tools/ablation-study.sh`) and the number of sessions (`TIMES`) can be reduced. To adapt for low memory and disk space, the batch size (`BATCH_SIZE`) can be reduced.
Execution: Run `sudo tools/ablation-study.sh`. It requires root privilege because a normal user might not have the permission to read the unzip outputs.
Results: The stats are saved in `evaluation/stats`. The samples and outputs are saved in the subdirectories of `evaluation/sessions`. Run `python3 tools/fuzz-stats.py evaluation/stats/*` to plot the graph (Fig. 4) at `inconsistent-pair-cdf.pdf`.

(E2): Ambiguous ZIP construction and inconsistency type counting [1 human-minute + 40 computer-minutes]
Execution: Inside the `zip-diff` directory, first run `target/release/construction`, and then run `sudo target/release/count`.
Results: The inconsistency type details are saved at `constructions/inconsistency-types.json`. Run `python3 tools/inconsistency-table.py` to generate the LaTeX parser inconsistency table (Table 4).

A.5 Notes on Reusability

More ZIP parsers can be added for testing. Refer to `01-infozip` as an example to configure the Docker image.

ZIPDIFF may also be adapted to test other archive formats, if the ZIP-level mutations are replaced with other format-specific ones.

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/usenixsec2025/`.