USENIX'23 Artifact Appendix: Reassembly is Hard: A Reflection on Challenges and Strategies

Hyungseok Kim[1,2], Soomin Kim[1], Junoh Lee[1], Kangkook Jee[3], and Sang Kil Cha[1]
[1]*KAIST* [2]*The Affiliated Institute of ETRI* [3]*University of Texas at Dallas*
{witbring,soomink,junoh,sangkilc}@kaist.ac.kr *kangkook.jee@utdallas.edu*

A Artifact Appendix

A.1 Abstract

REASSESSOR is a tool for finding errors in the implementations of existing reassemblers. This artifact includes the source code of REASSESSOR, the dataset used in our paper, and several scripts for reproducing the results in the paper. As a preprocessing step, one needs to run three existing reassemblers on our dataset, including Ramblr, RetroWrite, and Ddisasm. We provide a dockerized environment to run them. The preprocessing step produces a re-assemblable assembly file for each binary, and REASSESSOR uses those files to find reassembly errors. The next section details each step to reproduce the results in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Not applicable.

A.2.2 How to access

The source code of REASSESSOR is accessible through GitHub at `https://github.com/SoftSec-KAIST/Reassessor/tree/v1.0.0`. We also provide our dataset at `https://doi.org/10.5281/zenodo.7178116`.

A.2.3 Hardware dependencies

To reproduce the whole results, it requires at least 2.5TB of disk space. In our experiments, we used a machine equipped with 8 cores of CPUs and 128GB of RAM.

A.2.4 Software dependencies

REASSESSOR is designed to run on a Linux machine, and we tested it on Ubuntu 18.04 and Ubuntu 20.04. Also, REASSESSOR is written in Python 3 (3.6), and it depends on `pyelftools` (>= 0.29) and `capstone` (>= 4.0.2). Besides, Docker needs to be installed on the same machine to run reassemblers within a Docker container. Our scripts assume that you can run Docker commands as a regular (unprivileged) user; thus, no need to run them with `sudo`.

A.2.5 Benchmarks

Our benchmark is accessible through Zenodo: `https://doi.org/10.5281/zenodo.7178116`. We created our benchmark with various combinations of compilers, linkers, target ISAs, and compiler options.

- ISA: x86 and x86-64 (= 2)
- Compilers: GCC v7.5.0 and Clang v12.0 (= 2)
- PIE/non-PIE: produce a PIE or a non-PIE (= 2)
- Optimization: O0, O1, O2, O3, Os, and Ofast (= 6)
- Linkers: GNU ld v2.30 and GNU gold v1.15 (= 2)

Also, our benchmark was created by compiling two source packages totaling 122 executable programs as follows.[1]

- GNU coreutils (v8.30): 107 executable programs.
- GNU binutils (v2.31.1): 15 executable programs.

A.3 Set-up

A.3.1 Installation

Once you download our source code from the GitHub repository, you can install it using the following command:

```
$ pip3 install -r requirements.txt
$ python3 setup.py install -user
```

A.3.2 Basic Test

You can run REASSESSOR with a sample program as follows:
(Step 1): Build a sample program:

```
$ cd ./example
$ ./make
$ cd ..
```

[1]We exclude 31 programs in SPEC CPU 2006 from the dataset because of a licensing issue. Instead, we provide SSH server to grant access to all datasets we made.

(Step 2): Run `preprocessing.py` to reassemble it:

```
$ mkdir -p output
$ python3 -m reassessor.preprocessing \
./example/bin/hello ./output
$ ls output/reassem/
ddisasm.s retrowrite.s
```

(Step 3): Run REASSESSOR to find reassembly errors

```
$ python3 -m reassessor.reassessor \
example/bin/hello example/asm/ output/ \
--retrowrite ./output/reassem/retrowrite.s
$ ls output/errors/retrowrite/
disasm_diff.txt sym_diff.txt sym_errors.dat
sym_errors.json
```

REASSESSOR produces the following files as output: `ddisasm_diff.txt`, `sym_errors.dat`, `sym_diff.txt`, `sym_errors.json`. Firstly, `disasm_diff.txt` contains a list of disassembly errors (one per line); each line contains the relevant address, reassembler-generated assembly line, and compiler-generated assembly line. `sym_errors.dat` is a raw output file containing a list of symbolization errors. This file is used to generate other two files: `sym_errors.json` and `sym_diff.txt`. `sym_diff.txt` is a human-readable representation of `sym_errors.dat`. Each line of the file contains address, error type, reassembler-generated assembly code, and compiler-generated code, for each error found. Finally, `sym_errors.json` contains detailed information about each symbolization error found, including the relevant assembly file, line number, relocatable expression type, normalized code, repairability, and so on. The file is written in the JSON format.

A.4 Evaluation workflow

A.4.1 Preprocessing step for experiments

[10 human minutes + 5,000 CPU hours + 60 GB disk]

REASSESSOR finds reassembly errors by diffing compiler-generated assembly code and reassembler-generated assembly code. Thus, we need to reassemble benchmark binaries as follows:

(Step 1): Download the dataset:

```
$ cd artifact
$ tar -xzf /path/to/dataset/benchmark.tar.gz .
$ ls dataset/
binutils-2.31.1 coreutils-8.30
```

(Step 2): Run `run_preproc.py` to obtain reassembler-generated assembly code from each reassembler:

```
$ python3 run_preproc.py
```

`run_preproc.py` will then generate assembly files under the `./output` directory:

```
$ ls ./output
binutils-2.31.1 coreutils-8.30
```

```
$ cd \
output/binutils-2.31.1/x64/clang/nopie/o0-bfd/addr2line/
$ ls reassem
ddisasm.s ramblr.s
```

A.4.2 Major Claims

(C1): REASSESSOR *is able to find diverse reassembly errors. This is proven by the experiment (E1) described in Section 5.3 as well as Table 4.*

(C2): *Composite relocation expressions are prevalent in real-world binaries, and precise CFG recovery is a necessary condition for sound reassembly of x86-64 PIEs. This is proven by the experiment (E2) described in Section 5.2.2.*

(C3): *There are previously unknown FN/FP patterns. This is proven by the experiment (E3) described in Section 5.4.1 and 5.4.2.*

(C4): *Preventing data instrumentation can mitigate the symbolization challenge. This is proven by the experiment (E4) described in Section 5.5.2*

A.4.3 Experiments

(E1): [10 human minutes + 140 CPU hours + 330GB disk] The experiment will search for reassembly errors by running REASSESSOR.

How to: First, run `run_reassessor.py` to find reassembly errors. Second, run `classify_errors.py` to collect the errors. Third, run `get_summary.py` to get the summarized result.

Preparation: The preprocessing step in §A.4.1 is required to have reassemablable assembly files.

Execution: Run `run_reassessor.py`

```
$ python3 run_reassessor.py --core 6
```

Results: First, check the report files described in §A.3.2:

```
$ cd output/binutils-2.31.1/x64/clang/nopie/o0-bfd/
$ cd addr2line/
$ ls errors
ddisasm ramblr
$ ls errors/ddisasm/
disasm_diff.txt sym_diff.txt sym_errors.dat
sym_errors.json
```

Second, run `classify_errors.py` to collect and classify symbolization errors from `sym_diff.txt` files:

```
$ python3 classify_errors.py --core 8
```

Check the results under the `triage` folder:

```
$ ls triage
ddisasm ramblr retrowrite
$ ls triage/ddisasm/x64/nopie/
E1FN.txt E1FP.txt E2FN.txt E2FP.txt E3FN.txt
E3FP.txt ...
```

Each file has a different set of errors, and each line of the files contains a relevant file name, error type, reassembler-generated assembly line, and compiler-generated assembly line.

Third, run `get_summary.py` to get a summarized result presented in Table 4:

```
$ python3 get_summary.py --core 8
```

(E2): [10 human minutes + 2.5 CPU hours + 100MB disk] This experiment examines all relocatable expressions in our benchmark and reports the distributions of relocatable expressions for a different set of assembly files. Also, the experiment will show that the proportion of label-relative (Type VII) relocatable expressions in x86-64 PIE binaries is not negligible.

How to: First, run `get_asm_statistics.py` to examine compiler-generated assembly files. Second, run `get_e7_errors.sh` to find E7 errors for x86-64 PIE binaries.

Preparation: (E1) experiment needs to be run first since `get_asm_statistics.py` and `get_e7_errors.sh` refer to data files (`gt.dat` and `sym_diff.txt`) that REASSESSOR made.

Execution: Run `get_asm_statistics.py` and `get_e7_errors.sh`:

```
$ python3 get_asm_statistics.py -core 8
$ /bin/bash get_e7_errors.sh
```

Result: First, `get_asm_statistics.py` shows the distribution of relocatable expressions, and the proportion of composite relocatable expressions. Also, it reports how many binaries have abnormal cases including composite relocatable expressions pointing to outside of valid memory ranges and code pointers referring to non-function entries. Second, `get_e7_errors.sh` reports how many x86-64 binaries suffer from E7 errors.

(E3): [10 human minutes + 1 CPU minute + 2.2GB disk] This experiment will find previously unseen symbolization errors.

How to: Run `dissect_errors.sh` to find previously unseen symbolization errors.

Preparation: (E1) experiment is required since `dissect_errors.sh` examines symbolization errors from `sym_diff.txt` files.

Execution: Run `dissect_errors.sh`:

```
$ /bin/bash dissect_errors.sh
```

Results: `dissect_errors.sh` reports how many reassembler-generated files have previously unseen errors. Also, `dissect_errors.sh` generates the report files: `atomic_fn_cases.txt`, `atomic_fp_cases.txt`, and `label_err_fp_cases.txt`. Each line of the files contains a relevant file name, error type, reassembler-generated assembly line, and compiler-generated assembly line. `atomic_fn_cases.txt` contains false negative cases where reassemblers misidentify atomic relocatable expressions as literals. `atomic_fp_cases.txt` contains false positive cases where reassemblers falsely symbolize atomic relocatable expressions as composite forms. Lastly, `label_err_fp_cases.txt` contains cases where symbolized labels have the same form as in the original one, while only the label values are misidentified.

(E4): [10 human minutes + 1 CPU minute + 6GB disk] This experiment measures an empirical lower bound of the number of reparable symbolization errors when preventing data instrumentation. Specifically, this experiment will count symbolization errors that satisfy the criteria we suggested in Section 5.5.2.

How to: Run `check_reparable_errors.sh` to find symbolization errors that are reparable.

Preparation: (E1) experiment is required since `check_reparable_errors.sh` examines the error list files that `classify_errors.sh` generates.

Execution: Run `check_reparable_errors.sh`:

```
$ /bin/bash check_reparable_errors.sh
```

Results: `check_reparable_errors.sh` reports how many symbolization errors satisfy the reparable conditions we introduced in Section 5.5.2. Also, `reparable_errors.txt` contains the list of reparable symbolization errors; each line of the file has a relevant file name, error type, reassembler-generated assembly line, and compiler-generated assembly line.

A.5 Notes on Reusability

A.5.1 How to make a new dataset

If you want to use a new dataset, build binaries with '`-g`' option and '`--save-temps=obj`' option. Also, if you want to make non-pie binaries, add '`-Wl,--emit-relocs`' linker option to preserve relocation information.

A.5.2 How to test different versions of reassemblers

If you wish to run REASSESSOR with newer versions of reassemblers, update the execution commands in the `reassemble()` method in `preprocessing.py`.

A.6 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/.