Artifact for paper "Sound Abstract Nonexploitability Analysis"

Abstract

The goal of this artifact is to guide the user to reproduce the performance and precision evaluation of the paper presented in Section 7. In particular, we will analyze the Coreutils and Juliet benchmarks in order to reproduce the results reported in Table 1 of the paper. The static analysis framework that we use relies on low-level architecture-dependant details of Clang, so that the artifact should be tested on a x86-64 machine.

1 Getting Started

In this section, we first show how to setup the artifact, and then we show how to run some simple tests to verify that the analyzer runs correctly. The artifact includes a Docker image with all the dependencies already installed and configured.

Setup. With the artifact, we provide an image for `x86-64` to reproduce our experiments. To load the image, run the following command:

```
docker load < mopsa-artifact.tar.gz
```

Then, to start the container run the following:

```
docker run -it mopsa-artifact:latest
```

To run the image on architectures other than `x86-64`, it is necessary to use emulation:

```
docker run --platform=x86-64 -it mopsa-artifact:latest
```

The emulation is sensibly slower than the native execution. Observe that some systems require root privileges to run Docker. In this case, prefix the commands with `sudo`. The following sections will assume that the user is running the container. No additional setup is required, as we provide the container with the dependencies already installed and configured. The artifact comes with some utilities already installed, such as `vim` and `cloc`. To install a new package, one can run `sudo apt install {package}`. No password is required, and the default user has root privileges.

Structure of the container. The following directories are available:

- `mopsa-analyzer`: a modified version of the Mopsa analyzer.
- `exploitability-benchmarks`: directory containing Coreutils and Juliet benchmarks. There are two subdirectories, `coreutils` and `juliet`, containing the two test suites.
- `exploitability-examples`: a set of test programs to verify that the analyzer runs correctly.
- `apron`: source code for the Apron library, which is necessary to run our analyzer. We use the nightly version, so that we build the source code and we install it manually instead of using the OCaml package manager.

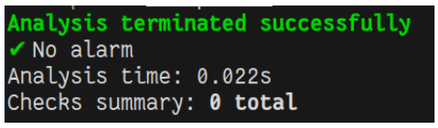
Basic testing. In this section, we describe how to quickly verify that the artifact is complete and runs correctly. After starting the Docker container, navigate to the `exploitability-examples` directory.

```
cd exploitability-examples
```

Now, we will analyze some programs to verify that the analysis runs correctly. The interested reader can inspect the contents of the files by running `cat {filename}`. First, run a classic analysis on the file `example1.c` with the following command:

```
mopsa-c example1.c
```

The analysis should terminate correctly, printing an output similar to the following:



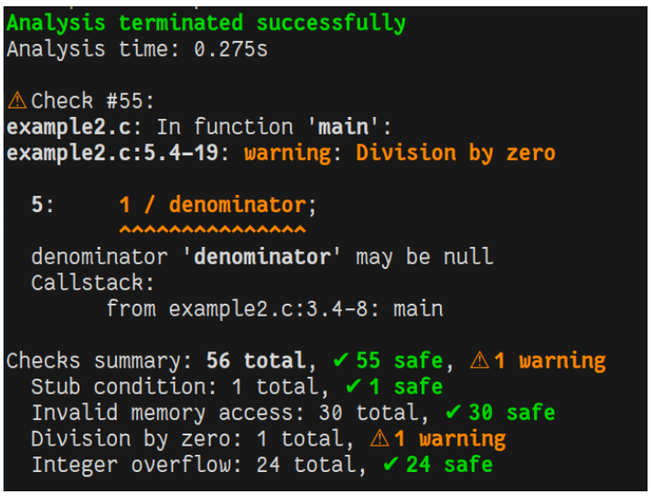
Run the exploitability analysis (Mopsa-Nexp) on the same program:

```
mopsa-nexp example1.c
```

Since the program is correct, the analysis output should be similar to the previous one. We now analyze `example2.c`, which is a program where there is a possible division by zero. Run the following command:

```
mopsa-c example2.c
```

Since we use the standard version of the analyzer, an alarm should be reported, saying that there is a possible division by zero:



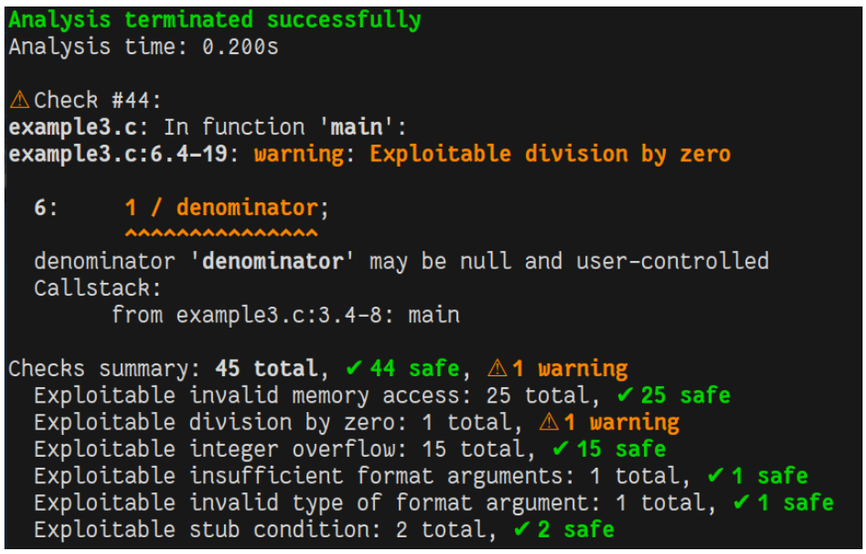
However, the division by zero cannot be triggered by an attacker (i.e. it is *nonexploitable*), as the denominator is not controlled by an external user. To verify that the nonexploitability analysis does not classify this program as exploitable, run the following command:

```
mopsa-nexp example2.c
```

The analyzer should not report any alarm, as the division by zero cannot be triggered by an attacker. As last example, run the exploitability analysis on a program that has an *exploitable* division by zero. Run the following command:

```
mopsa-nexp example3.c
```

The output should indicate that the denominator might be null and user-controlled:



If all the steps were executed without errors, then both Mopsa and Mopsa-Nexp are installed correctly. We now run a small automated test suite to verify that it is possible to correctly analyze the benchmarks from Coreutils and Juliet. Run the following commands:

```
cd /home/mopsa-user/exploitability-benchmarks
make tests
```

This will run a small test suite analyzing a program from Coreutils and some test cases for Juliet. If the command runs without errors, this confirms that the artifact is complete, and that it is possible to run the full test suite to reproduce the experimental results. Observe that the command can output some warnings while compiling the Coreutils programs.

2 Replicating the experimental evaluation

In this section, we describe how to reproduce the experimental results that appear in the paper. In particular, we show how to reproduce the experiments that we discuss in Section 7, subsection "Performance and Precision Evaluation" (see pages 16-18 of the paper).

Replicable results. Our objective is to run the analyzer on the Coreutils and Juliet benchmarks, and to reproduce the results shown in Table 1 (see page 17 of the paper). The *precision* results (that is, the number of alarms shown in the column **Alarms**) should be *fully reproducible*. The *performance* results (that is, the runtime shown in the column **Time**) *vary depending on the machine* on which the experiments are executed. Nevertheless, it should be possible to observe:

- A substantial performance overhead when comparing Mopsa-Nexp with Mopsa for Coreutils. In our case, the performance overhead was around 16%, but this can vary.
- A small performance overhead when comparing Mopsa-Nexp with Mopsa for Juliet. In our case, the performance overhead was around 2%, but this can again vary. As the programs in the Juliet test suite are very small, we believe it would even be possible to observe a slightly negative overhead, even though this never occurred in our experiments.

Resources requirements. As described in the paper, we ran our experiments on a server with 128GB of RAM, 48 Intel Xeon CPUs E5-2650 v4 @ 2.20GHz and Ubuntu 18.04.5 LTS. With this setup, it takes around 2 hours to run the whole test suite. However, *it is not necessary to have such a powerful machine* to run the experiments. In fact, we provide a smaller version of the experiments that can be executed on a

regular machine in less than 30 minutes. We tested this smaller version on a computer with 8 Intel Core CPUs i7-8650U @ 1.90GHz, 16GB of RAM and Ubuntu 18.04.5 LTS, and we were able to run the experiments in 13 minutes.

Running the experiments. We provide scripts to run all the experiments and produce output data. The results are saved as JSON files, and can be analyzed later. The scripts that we provide automatically parallelize the execution of the experiments. First, navigate to the `exploitability-benchmarks` directory:

```
cd /home/mopsa-user/exploitability-benchmarks
```

To run the *full* benchmarks, run the following command:

```
make benchmark
```

On a machine with eight cores this can possibly take more than 8 hours. For this reason, we provide a reduced version of the benchmarks that on a regular machine with eight cores should run under 30 minutes. To run the experiments on the reduced version of the benchmarks, run the following alternative command:

```
make benchmark-fast
```

The commands first compile the Coreutils programs, and then they run the experiments on the benchmarks. The results are saved in the subdirectories `coreutils/results` and `juliet/results`. In case the execution of the experiments is interrupted by pressing `Ctrl-C`, to delete corrupted data in the results run `make clean`. The JSON reports of the analyses can be manually inspected, and they contain human-readable information about the alarms raised by the analyzer. Nevertheless, they are up to 159,600 reports. We provide scripts to automatically analyze the data and pretty-print formatted results. In the next section, we describe how to use these scripts.

Analyzing the results. The subdirectory `scripts` contains the file `all-stats.py`, which accepts as input the path that contains the experiments' results. To analyze the results for Coreutils run the following:

```
./scripts/all-stats.py -i coreutils/results
```

It might take a few seconds to print the results, as there are many files to open and analyze. The script prints three tables, one for each abstract domain that we considered (intervals, octagons, and polyhedra). The first table corresponds to the first two rows in Table 1 (not counting the header), the second table corresponds to the third and fourth rows, and the third table corresponds to the fifth and sixth rows. As previously mentioned, if considering the full benchmarks, the number of alarms should be fully reproducible. The performance results can largely vary depending on the machine used to run the experiments. The scripts also print, for each abstract domain, the absolute number of warnings proved nonexploitable, the percentage of warnings proved nonexploitable, and the exploitability analysis performance overhead. Observe that these numbers can be trivially derived from the data in the tables. For the reduced version of the benchmarks, for Coreutils we obtained the following results:

Domain	**Analyzer**	**Alarms**	**Time**
Intervals	Mopsa	640	0:07:08
	Mopsa-Nexp	164	0:08:15
Octagons	Mopsa	632	0:13:24
	Mopsa-Nexp	163	0:14:41
Polyhedra	Mopsa	632	0:13:21
	Mopsa-Nexp	163	0:14:41

Observe that with fewer examples, the performance results become more unpredictable. Nevertheless, the reader should be able to observe similar performance trends. To analyze the results for Juliet run the following:

```
./scripts/all-stats.py -i juliet/results
```

The three tables printed by the previous command correspond to the last six rows in Table 1. For the reduced version of the benchmarks, for Juliet we obtained the following results:

Domain	Analyzer	Alarms	Time
Intervals	Mopsa	279	0:03:16
	Mopsa-Nexp	80	0:03:22
Octagons	Mopsa	270	0:03:42
	Mopsa-Nexp	73	0:03:50
Polyhedra	Mopsa	270	0:03:38
	Mopsa-Nexp	73	0:03:45

3 Additional information

In this section we report additional information about the artifact that is not necessary to reproduce the experimental results, but that can be useful for the interested reader.

Mopsa and Mopsa-Nexp. The source code of the analyzer is in the directory `mopsa-analyzer`. Mopsa is a complex analyzer, and fully describing its architecture and its capabilities is beyond the scope of this guide. We refer the interested reader to [3] for an in-depth overview of the design principles of the analyzer. Furthermore, the Mopsa user manual is available online at [4]. The user manual shows how to fully exploit Mopsa's capabilities to analyze complex C and Python programs. The manual also describes the large set of available command line options, which can be listed with `mopsa -help`. The source code of the analyzer contains extensive documentation, and it can be found in the `mopsa-analyzer/analyzer` directory:

- The core of the analyzer is defined in `framework`.
- The analysis for Python is implemented in `languages/python`, but it is out of the scope of this artifact.
- The analysis for C is implemented in `languages/c`.
- The analyses common to C and Python, such as loop invariant inference and some intraprocedural constructs is implemented in `languages/universal`.
- The nonexploitability analysis is mainly implemented in `languages/c/taint`. In this directory, the interested reader will find modified versions of various abstract domains, such as cells [2] (`cells.ml`), pointers (`pointers_tainted.ml`), and machine numbers (`machine_numbers.ml`). These three domains are the most interesting ones, as they are those that track the taint information.

The analyzer allows the user to compose the abstract domains in different configurations specified as JSON files. The exploitability analysis is simply a C analysis that relies on modified versions of some abstract domains, and the analyzer provides a set of configurations to run the nonexploitability analysis.

Analyzing additional programs. The command `mopsa-nexp` runs the exploitability with the interval abstract domain. However, it is possible to use different underlying abstract domains. The following illustrates how to run various types of analyses on a C program.

```
# Nonexploitability analysis using intervals.
# mopsa-nexp is an alias to the following.
mopsa-c -config=c/cell-itv-taint.json {FILE.c}

# Nonexploitability analysis using octagons.
mopsa-c -config=c/cell-pack-rel-itv-taint.json \
        -numeric=octagon {FILE.c}
```

```c
#include <stdio.h>

int main() {
    int x;
    scanf("%d", &x);
    x = x % 10;
    int y = x;
    int denominator = x - y;
    1 / denominator;
    return 0;
}
```

Figure 1: C program with a nonexploitable division by zero

```
# Nonexploitability analysis using polyhedra.
mopsa-c -config=c/cell-pack-rel-itv-taint.json \
        -numeric=polyhedra {FILE.c}

# Nonexploitability analysis using linear equalities.
mopsa-c -config=c/cell-pack-rel-itv-taint.json \
        -numeric=lineq {FILE.c}
```

Observe that the relational domains use a technique called *packing* [1] (hence, `pack` in the configuration names) to be more efficient. The interested reader can analyze additional programs with different configurations. For instance, it would be interesting to analyze the program represented in Figure 1. Even if the denominator is assigned to an expression that contains a variable that is user-controlled, it will always be zero. This is implies that the denominator is not tainted, and hence the program is nonexploitable accordingly to our formal definition of exploitability. A simple interval analysis cannot infer this information, as it is not precise enough. By comparing the output of the interval analysis with a relational analysis (for instance, using polyhedra), it is possible to observe that the latter can prove the program to be nonexploitable, while the former cannot. Observe that MOPSA can analyze also complex programs that are compiled using different build systems such as `make` or `cmake`. We refer the reader to the MOPSA user manual [4] for this advanced use case.

References

[1] P. Cousot et al. "The Astrée analyzer". In: *European Symposium on Programming ESOP*. Vol. 3444. Lecture Notes in Computer Science (LNCS). `http://www-apr.lip6.fr/~mine/publi/esop05_astree.pdf`. Springer, 2005, pp. 21–30. DOI: `10.1007/978-3-540-31987-0_3`.

[2] A. Miné. "Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics". In: *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*. `http://www-apr.lip6.fr/~mine/publi/article-mine-lctes06.pdf`. ACM, 2006, pp. 54–63.

[3] M. Journault et al. "Combinations of reusable abstract domains for a multilingual static analyzer". In: *Proc. of the 11th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE19)*. Vol. 12031. Lecture Notes in Computer Science (LNCS). `http://www-apr.lip6.fr/~mine/publi/article-mine-al-vstte19.pdf`. Springer, 2019, pp. 1–18. DOI: `10.1007/978-3-030-41600-3_1`.

[4] *MOPSA User Manual*. URL: `https://mopsa.gitlab.io/mopsa-manual/user-manual/`.