USENIX Security '24 Artifact Appendix: Inference of Error Specifications and Bug Detection Using Structural Similarities

Niels Dossche
Ghent University

Bart Coppens
Ghent University

A Artifact Appendix

A.1 Abstract

Error-handling code is a crucial part of software to ensure stability and security. We propose a novel approach to automatically infer error specifications for system software without a priori domain knowledge, while still achieving a high recall and precision, and leverage this information to find missing error checks, incorrect error checks, and error propagation bugs. We implemented this approach in a tool called ESSS. In our evaluation, we demonstrate the effectiveness and efficiency of our approach on 7 well-tested, widely-used open-source software projects: OpenSSL, OpenSSH, PHP, zlib, libpng, freetype2, and libwebp. First, we show that ESSS is scalable with regards to both computation time as well as memory usage. Then, we show that the inferred error specifications are more precise than those inferred by the prior state of the art, EESI. Then, we evaluate the effectiveness of our tool to find bugs. On the aforementioned open source projects, our tool reports 827 potential bugs in total for all 7 projects combined. We manually categorised these 827 issues into 279 false positives and 541 true positives. Finally, we evaluate false negatives on the APIMU4C dataset, and compare against CodeQL and APISan.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no such concerns for the evaluators or their machines.

A.2.2 How to access

The artifact containing everything, including all benchmarks, our own tool, CodeQL, EESI, and APISan (all where necessary patched to make them work, these patches are also included); all pre-built, is distributed as a (VirtualBox) Virtual Machine image, which is available on Zenodo at https://zenodo.org/doi/10.5281/zenodo.10843435. The version used during evaluation is available at https://doi.org/10.5281/zenodo.11118948.

Important: If needed, the username and password of the VM user are both `evaluation`.

If people want to use our tool outside of the VM: the source code of our own tool for this artifact is also included separately in the Zenodo dataset outside of the VM (it is, of course, also included in the VM itself). The version used for the artifact evaluation is available at https://github.com/csl-ugent/ESSS/tree/b77210f82654dbbd8f50aea3fe3216902c86ccd1.

The ESSS repository also contains the script necessary to build the VM image. It is located at `vm/build_vm.sh`. This script reproduces the VM and should be executed on a clean Ubuntu 22.04 LTS installation.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

VirtualBox for the virtual machine image. The virtual machine image contains all dependencies necessary to compile and evaluate our tool ESSS, as well as all dependencies for tools and benchmarks we compared and evaluated against. If you want to compile our tool ESSS yourself, you need a basic Linux install with build-essentials and CMake. We tested this on Ubuntu 22.04.

A.2.5 Benchmarks

The benchmarks we used to evaluate ESSS and EESI are:

- OpenSSL (commit 8d927e55)
- OpenSSH (commit 36c6c3ef)
- PHP (commit abc41c2e)
- zlib (commit 12b345c4)
- libpng (commit e519af8b)
- freetype2 (commit bd6208b7)
- libwebp (commit 233960a0)

We also used the APIMU4C bug dataset, which also contains the following benchmarks that the authors of that dataset have patched to reintroduce some bugs:

- OpenSSL (1.1-pre8)
- httpd 2.4.37
- curl 7.63.0

The APIMU4C dataset contains a patch to make curl compilable, because the authors used C++-style comments to mark certain parts of the code which makes compilation not pass curl's linter. The patch file is provided at `APIMU4C/APIMU4C/curl-curl-7_63_0_patched/curl-curl-7_63_0/patch`

We additionally pre-compiled musl 1.2.3 in the VM, inside the `build-musl-1.2.3` directory in the home directory.

A.3 Set-up

We have prepared an environment in the VM that has all the necessary dependencies and configurations already applied.

A.3.1 Installation

The VM has all the software precompiled and dependencies installed. All listed paths are relative to the home directory.

Only if you wish to compile and/or install the tools manually, please perform the following instructions.

ESSS For ESSS, go to the `ESSS/analyzer` directory and run the `make` command. The only dependency is LLVM 14.0.6, which can be built using the `ESSS/llvm/build-llvm.sh` script. The built binary will be located at `ESSS/analyzer/build/lib/kanalyzer`.

EESI For EESI, the source code is checked out in `tools/eesi`. The upstream code is available at https://github.com/ucd-plse/eesi/tree/aacbcecbb8c1f1d2e152ee4a0aa417ac92ae74bb (and is also included in the VM). Note that this source code is patched to make it compatible with LLVM 14.0.6, and to fix a crash bug we encountered. The patch file is provided at `tools/eesi/0001.patch`. The patched code can be obtained directly from https://github.com/nielsdos/eesi-updated/tree/687180d496dadb9110bbfeddb0b5353341f4933b. You can build it by performing the following steps:

- Create a build directory `build` in `tools/eesi`
- Set the environment variable `CMAKE_PREFIX_PATH` to `/home/evaluation/ESSS/llvm/llvm-project/prefix`
- Run `cmake ../src` in your build directory

The binary `eesi` will be placed in your build directory.

APISan For APISan, we use a Dockerfile that we made ourselves based on a patched version to make it compatible with the Ubuntu 18.04 that the Docker container uses. The upstream code is available at https://github.com/sslab-gatech/apisan/tree/9ff3d3bc04c8e119f4d659f03b38747395e58c3e (and is also included and patched in the VM). The patch file is provided at `deploy/apisan/0001.patch` and the Dockerfile is located in `deploy/apisan`. The patched code can be obtained directly from https://github.com/nielsdos/apisan/tree/15b697819610b4dd0671c8f420a552dbf0a46e04. We prebuilt the Docker container with the name `apisan`.

CodeQL CodeQL (version 2.13.3) is downloaded as-is from https://github.com/github/codeql-action/releases/tag/codeql-bundle-20230524. This is checked out in `tools/codeql`.

Benchmarks The instructions to compile each benchmark are included in the `ESSS/evaluation/benchmark-instructions` directory. The benchmarks are already precompiled in the VM.

A.3.2 Basic Test

To run a basic smoke test, please follow these instructions:

ESSS Run `ESSS/build/lib/kanalyzer` which should print the usage of the tool.

EESI Run `tools/eesi/build/eesi` which should print the usage of the tool.

APISan Run `docker run --rm -it apisan bash` followed by `/apisan/apisan` which should print the usage of the tool.

CodeQL Run `codeql` which should print the usage of the tool.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our tool ESSS infers more specifications, and has a higher precision and recall than the state-of-the-art EESI tool. Furthermore, it does so in less time and uses less memory. This corresponds to Table 1 and Table 2 in the paper. This is proven by experiments (E1) and (E2): (E1) has EESI infer specifications from the benchmarks, while keeping track of its execution time and memory usage; (E2) has our own tool ESSS infer specifications from the benchmarks, again while keeping track of its execution time and memory usage.

(C2): ESSS has been used to detect new bugs in all 7 benchmarks (OpenSSL, OpenSSH, PHP, zlib, libpng, freetype2, libwebp). We categorised these bug reports ourselves into false positives, true positives, and unknowns. This corresponds to Table 3 in the paper. This is proven by experiment (E3).

(C3): Evaluating ESSS, APISan, and CodeQL on the APIMU4C dataset, shows that ESSS outperforms the prior art in number of bugs found in almost all cases. Furthermore, it does so in less time and uses less memory. This corresponds to Table 4 in the paper. This is proven by experiment (E4).

(C4): Of all the bugs ESSS detected, we submitted patches for 46 of them, which were all confirmed by the developers and accepted. We evaluated how well APISan and CodeQL are able to find these 46 bugs. This corresponds to Table 5 in the paper. This is proven by experiment (E5).

A.4.2 Experiments

(E1): *EESI specifications [1 human-hour + 100 compute-minutes (for 10 runs) + 22GiB RAM]: This experiment reproduces the specifications for the state-of-the-art existing tool EESI.*

How to: *We need to run EESI for all benchmarks, except PHP for which it goes out of memory even of a machine with 128 GiB memory. A single run will take about 10 compute-minutes. We used 10 runs to obtain the time and memory statistics.*

Preparation: *Go to the* `ESSS/evaluation` *directory. This contains the scripts necessary to evaluate both EESI and ESSS.*

Execution: *The evaluation directory contains a script* `run-eesi-<program>.sh` *for each program. Execute them all (except for PHP, which will not finish as described above). This will output to stdout the error specifications inferred by EESI and the time it took and the memory usage in KiB. We recommend saving the results to a file for each benchmark.*

Results: *You can average the time and memory usage over the 10 runs to obtain the statistics. When we evaluated the specifications, we randomly sampled using the* `random_sampling_of_lines.py` *script. The annotated precision (prefixed with T (true) / F (false)) is in the file* `eesi-<program>-precision`*. The sample for recall (for both EESI and ESSS) is in* `<program>-recall-sample`*. Our output from EESI (with the memory and time stripped) is located in* `eesi-<program>-output`*, and this file will be used by the statistics script. You can compute the statistics using* `compute_eesi_stats.py eesi-<program>-output eesi-<program>-precision <program>-recall-sample`*.*

(E2): *ESSS specifications [1 human-hour + 10 compute-minutes + 2GiB RAM]: This experiment reproduces the specifications for our tool ESSS.*

How to: *We need to run ESSS for all benchmarks. A single run will take about 1 compute-minute. We used 10 runs to obtain the time and memory statistics.*

Preparation: *Go to the* `ESSS/evaluation` *directory.*

Execution: *The evaluation directory contains a script* `run-my-<program>.sh` *for each program. Execute them all. This will output to stdout the error specifications (and bug reports) inferred by ESSS and the time it took and the memory usage in KiB. We recommend saving the results to a file for each benchmark.*

Results: *You can average the time and memory usage over the 10 runs to obtain the statistics. When we evaluated the specifications, we randomly sampled using the* `random_sampling_of_my_specs.py` *script. The ground truth for precision is for a subset sampled of ESSS, available in the file* `<program>-precision-ground-truth`*. The random sampling itself is located in the file* `<program>-random-functions-for-precision-my-tool`*. The sample for recall (for both EESI and ESSS) is in* `<program>-recall-sample`*. Our output from ESSS (with the memory and time stripped) is located in* `esss-<program>-output`*, and this file will be used by the statistics script. You can compute the statistics using* `compute_my_stats.py <program>`*.*

(E3): *ESSS found bugs [1 human-hour + 10 compute-minutes]: This experiment reproduces the bug reports for our tool ESSS.*

How to: *We need to run ESSS for all benchmarks. If you have saved the results from E2 you can reuse those files.*

Preparation: *Go to the* `ESSS/evaluation` *directory.*

Execution: *The evaluation directory contains a script* `run-my-<program>.sh` *for each program. Execute them all. This will output to stdout the bug reports inferred by ESSS. We recommend saving the results to a file for each benchmark as* `my-<program>-output`*.*

Results: *If you saved the output as the recommended file name, you can execute* `check_found_bugs.py <program>` *to get the statistics of found bugs. This uses the* `<program>-bugs` *file to check for bugs. That file contains for each filename and line number the category to which the reports belongs. The header of that file described each category.*

(E4): *APIMU4C evaluation [2 human-hours + 5 compute-hour + 27 GiB RAM]: This experiment reproduces the results for APIMU4C.*

How to: *We need to run CodeQL, APISan, ESSS for all 3 APIMU4C benchmarks (curl, OpenSSL, httpd).*

Preparation: *The APIMU4C benchmarks are in the* `APIMU4C/APIMU4C/` *directory.*

For each benchmark in APIMU4C, go inside the source directory of said benchmark and create a CodeQL database using `codeql database create --language=cpp -- db`*.*

To run (and prepare) APISan, go inside the source directory and run `docker run -it --rm -v "$(pwd)":/mnt apisan bash`. *The next commands take place in the Docker container. Then run* `/apisan/apisan build -- ./configure [options]` *using the configuration options and instructions as described in* `ESSS/evaluation/benchmark-instructions`. *For example, to build httpd you should follow the instructions in* `ESSS/evaluation/benchmark-instructions/httpd-instructions`. *This will require installing dependencies that are requested by the configure script. Dependencies can be installed using* `apt install liblua5.1-dev pkg-config libjansson-dev libssl-dev libcurl4-openssl-dev time uuid-dev`. *Then run* `/apisan/apisan build -- make`.
The source code is already precompiled for ESSS.
Execution: *To run the tests for CodeQL, execute* `codeql database analyze db --format=csv --output=<filename> <query>` *for each of the 3 benchmarks. We recommend running it twice because the first time a query executes, it needs to be compiled. The* `<query>` *parameter is the path to the query file. The queries we used in our evaluation are:* `MissingNullTest`, `ReturnValueIgnored`, `InconsistentNullnessTesting`, `InconsistentCheckReturnNull`, `MissingNegativityTest`.
To run the tests for APISan, execute `/apisan/apisan check --checker=rvchk` *for each of the 3 benchmarks. This will output the bug reports to stdout. Save the results to a file.*
To run the tests for ESSS, execute `ESSS/evaluation/run-my-<program>-apimu4c.sh` *for each of the 3 benchmarks.*
For all tests, you can use `/usr/bin/time -v` *to measure memory and run time.*
Results: *The results need to be checked against* `APIMU4C/APIMU4C/bug-location.xlsx` *that correspond to the missing or incorrect error check cases (the spreadsheet contains other error types too).*

(E5): *CodeQL, APISan, and ESSS for new bugs [2 human-hours + 16 compute-hours + 60 GiB RAM]: This evaluates CodeQL, APISan, and ESSS on the 46 bugs we submitted patches for.*
How to: *We need to run CodeQL and APISan for OpenSSL, OpenSSH, and PHP. The results for ESSS were already obtain in an earlier experiment, and as such no extra steps are needed for ESSS.*
Preparation: *Follow the same steps as in (E4) to prepare the OpenSSL and OpenSSH benchmark for CodeQL and APISan. In contrast to (E4), we will not use the version from APIMU4C, but we will use the version on which we evaluated ESSS. These are available in the* `benchmarks` *directory in the home directory.*
Execution: *Follow the same steps as in (E4) to run APISan and CodeQL for the OpenSSL, OpenSSH, and PHP benchmarks. For CodeQL, we use the same queries as in (E4).*
For all tests, you can use `/usr/bin/time -v` *to measure memory and run time.*
Results: *Compare the bug reports that are outputted against the bug reports from Table 6 in the paper's Appendix. For ESSS, you can use the* `check_found_bugs.py` *script as in (E3).*

A.5 Notes on Reusability

None.

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/usenixsec2024/.