


CtChecker: A Precise, Sound and Efficient Static Analysis for Constant-Time Programming (Artifact)

Quan Zhou ✉ 

Penn State University, University Park, PA, USA

Sixuan Dang ✉ 

Duke University, Durham, NC, USA

Danfeng Zhang ✉ 

Duke University, Durham, NC, USA

Abstract

This artifact includes the implementation of the CtChecker analysis toolchain described in the corresponding paper. We provide two options to run CtChecker, building it from source or running the

pre-built tool with Docker. All evaluated benchmark source code are provided in the artifact. A walkthrough of how to reproduce the evaluation results in the paper is provided in the Appendix.

2012 ACM Subject Classification Security and privacy → Information flow control

Keywords and phrases Information flow control, static analysis, side channel, constant-time programming

Digital Object Identifier 10.4230/DARTS.10.2.26

Related Article Quan Zhou, Sixuan Dang, and Danfeng Zhang, “CtChecker: A Precise, Sound and Efficient Static Analysis for Constant-Time Programming”, in 38th European Conference on Object-Oriented Programming (ECOOP 2024), LIPIcs, Vol. 313, pp. 46:1–46:26, 2024.

<https://doi.org/10.4230/LIPIcs.ECOOP.2024.46>

Related Conference 38th European Conference on Object-Oriented Programming (ECOOP 2024), September 16–20, 2024, Vienna, Austria

Evaluation Policy The artifact has been evaluated as described in the ECOOP 2024 Call for Artifacts and the ACM Artifact Review and Badging Policy.

1 Scope

This artifact is released as a Docker image that includes a pre-built CtChecker. All the benchmark and evaluation scripts are provided inside the image to help reproduce the evaluation results presented in the paper.

2 Content

The artifact package includes:

- Source code for CtChecker.
- Evaluation scripts and benchmarks described in the corresponding paper.

The detailed organization of the artifact is provided in Appendix A.

3 Getting the artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS). The artifact is available as a Docker image at: <https://hub.docker.com/r/ctchecker/ctchecker>. In addition, the source code and future development of the project is available at: <https://github.com/psuplus/CtChecker>.



© Quan Zhou, Sixuan Dang, and Danfeng Zhang;
licensed under Creative Commons License CC-BY 4.0
Dagstuhl Artifacts Series, Vol. 10, Issue 2, Artifact No. 26, pp. 26:1–26:5



DAGSTUHL
ARTIFACTS SERIES
Schloss Dagstuhl – Leibniz-Zentrum für Informatik,
Dagstuhl Publishing, Germany



4 Tested platforms

We've been building the system under Ubuntu 20.04, which provides the best overall compatibility for the toolchain. Other versions of Linux distributions as well as Mac OS may also work, but sometimes require extra user effort in setting up the compilation environment for a successful building process.

5 License

The artifact is available under the MIT License.

6 MD5 sum of the artifact

007ae89b130203f18cde0c467617a72e

7 Size of the artifact

218 MiB

A Directory Layout

The detailed organization of the artifact is shown in Figure 1, where the numbers represent directory levels.

```

0 PROJECT_ROOT
  1 projects                                # CtChecker root directory
    2 llvm-deps                             # our information flow analysis
      3 include                             # header files
      3 lib                                 # source code for the analysis
      3 mod_exp_tests                       # evaluations for the paper
      4 BearSSL0.6                         # BearSSL source code and scripts
      4 ct-rewriter-files                   # benchmarks for rewrites
        5 Constantine                      # rewritten code for Constantine
          6 results.csv                    # Constantine results
        5 SC-Eliminator-original           # benchmarks for SC-Eliminator
          6 results-ctchecker.csv         # SC-Eliminator results
      4 ct-verif-files                     # ct-verif comparison benchmarks
      4 libgcrypt1.10.1                   # Libgcrypt source code and scripts
      4 mbedtls3.2.1                      # mbedTLS source code and scripts
      4 openssl_1_1_1q                    # OpenSSL source code and scripts
      4 results                            # the results folder for crypto-libs
    3 processing_tools                     # scripts to automate the workflow
    3 sensitivity-tests                    # unit-tests
  2 poolalloc                             # DSA points-to analysis
  1 ...                                    # llvm infrastructure

```

■ **Figure 1** The detailed organization of the artifact.

B Using CtChecker

B.1 Building CtChecker from Source

1. Before building, make sure to check:
 - The default 'python' is linked to a python2 executable. Check by `python --version`.
 - gcc-9 (preferably) is installed for compiling LLVM 3.7.1. Use `gcc -v` to check version.
 - Use `update-alternatives` to change default `python` and `gcc` of the system if versions do not match.
2. Clone the project.
3. To build the toolchain, run the commands as in Figure 2.

```
# First direct to project's root dir
cd /PATH_TO_LLVM_DIR

# Configure the project under root and run 'make' to build LLVM
./configure
make

# Direct to projects folders, configure and make for each package.
cd projects/poolalloc/
./configure
make

cd ../llvm-deps/
./configure
make
```

■ **Figure 2** Build script

B.2 Running Pre-built CtChecker with Docker

Make sure Docker has been correctly installed on the test machine. The docker image is available on DockerHub. Get the container running with the following commands as shown in Figure 3.

```
# Pull Docker image from DockerHub
docker pull ctchecker/ctchecker:latest

# Run a container with the image
docker run --name ctchecker -dit ctchecker/ctchecker

# Get into the container's bash
docker exec -it ctchecker bash
```

■ **Figure 3** Get CtChecker from Docker

B.3 Running the Cryptographic Library Benchmark

If CtChecker is built from source, `PATH_TO_LLVM_DIR` refers to the root directory of the source code. For the container, it refers to `/artifact/ctchecker`. The cryptographic library benchmarks can be run as shown in Figure 4.

26:4 CtChecker (Artifact)

```
# Direct to the benchmark folder
cd /PATH_TO_LLVM_DIR/projects/llvm-deps/mod_exp_tests

# Running the analysis
# This script runs all four crypto libraries and
# their variations for comparison with ct-verif
./runall.sh
```

■ **Figure 4** Running the crypto-lib benchmarks

The results for this benchmark are generated under the directory `.../mod_exp_tests/results`, where four sub-folder will be created. The full folder is for full source versions, `min` for the minimal source versions, `ct_verif_files` for `ct-verif`'s minimal source code versions, and `ct_verif_files_full` for `ct-verif`'s full source code version.

B.4 Running the Benchmark on Rewritten Code by Constantine

Run Constantine [1] benchmarks as shown in Figure 5.

```
# Direct to the benchmark folder
cd /PATH_TO_LLVM_DIR/projects/llvm-deps/mod_exp_tests/
cd ct-rewriter-files/Constantine

# Running the analysis
# This script runs all algorithms that are
# successfully translated back to C source file
./test.sh
```

■ **Figure 5** Running Constantine benchmarks

The results for Constantine rewritten code are located under each algorithm's own folder. The aggregated result will be created under Constantine root folder with name `results.csv`.

B.5 Running the Comparison with SC-Eliminator on Their Benchmarks

Run SC-Eliminator [2] benchmarks as shown in Figure 6.

```
# Direct to the benchmark folder for SC-Eliminator
cd /PATH_TO_LLVM_DIR/projects/llvm-deps/mod_exp_tests/
cd ct-rewriter-files/SC-Eliminator-original

# Running the analysis
# This script runs benchmarks in SC-Eliminator paper
./test.sh
```

■ **Figure 6** Running SC-Eliminator benchmarks

The results are collected under the `SC-Eliminator-original` folder, in the file named `results-ctchecker.csv`.

References

- 1 Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 715–733, 2021.
- 2 Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 15–26, 2018.