

# Rumpsteak artifact

## 1. Content of the artifact

The artifact contains the following files (TODO: add source repository):

```
.  
|-- Artifact.md  
|-- Artifact.pdf  
|-- Dockerfile  
|-- gen_fig_6.sh  
|-- gen_fig_7.sh  
`-- getting_started.sh
```

Note that an internet connection is required to use the artifact.

## 2. Getting started guide (Docker artifact)

In this section, we will run the benchmarks used to produce Fig. 6 and Fig. 7 of the paper in a Docker container. At the end of this section, you should be have all the plots of those figures.

Note that the benchmark results may not match those in Fig. 6 and Fig. 7 when run in a Docker container or a machine with different specification than used in the paper. See the section on *Claims supported or not by the artifact* for more discussion of this.

To run the getting started guide, we assume you are running a Linux machine with Docker and Gnuplot installed, as well as standard Unix tools (`tail`, `awk`, `cut`, etc.).

This *Getting started* section is fully automated: extract the archive and run the `getting_started.sh` script. *The script takes a long time to run all the benchmarks. On the author's laptop, it took approximately 2 hours and 15 minutes.* When the script finishes, you should have a few `*.png` plots which correspond to the plots shown in Fig. 6 and Fig. 7 of the paper.

```
$ cd /path/to/extracted/artifact/  
$ ./getting_started.sh
```

Once run, to clean-up your system, in addition to removing the archive folder, you should remove the docker image (the following command assumes you don't have other docker images/containers on your system):

```
$ docker rmi rumpsteak_tool  
$ docker system prune
```

## 3. Step-by-step instructions

In this step-by-step section, we will see how to



- run individual examples, including custom-written ones;
- run the benchmarks and add new benchmarks; and
- explore the raw results of the benchmarks run in the *Getting started* section.

### 3.1. Prerequisites

The following software needs to be installed on the benchmark machine:

- Haskell and Cabal
- OCaml (tested on 4.11.2) and opam
- Rust (both stable 1.54.0 and nightly-2021-07-06) and Cargo
- Gnuplot and standard Unix tools
- libssl-dev (check that `pkg-config openssl` does not fail)

### 3.2. Installing third-party software

This subsection shows the commands to run in order to install the tools we compare Rumpsteak with, and third party tools. Alternatively, you can log into the docker machine built in the *Getting started* section (`docker run -ti --name Artifact rumpsteak_tool:latest`). Using a Docker container may affect the performance.

#### 3.2.A. Installing *k*-MC (CAV '19)

```
$ git clone https://bitbucket.org/julien-lange/kmc-cav19.git
$ cd kmc-cav19
$ ghc KMC -threaded
```

Then, copy (or link) the KMC binary in your `$PATH` under the name `kmc`.

#### 3.2.B. Installing SoundBinary

```
$ git clone https://github.com/julien-lange/asynchronous-subtyping.git
$ cd asynchronous-subtyping/tool
$ ghc Checker
```

Then, copy (or link) the Checker binary in your `$PATH` under the name `concur19`.

#### 3.2.C. Installing NuScr

```
$ opam install nuscr
```

#### 3.2.D. Installing Hyperfine

```
$ cargo install hyperfine
```



### 3.3. Installing Rumpsteak

Rumpsteak consists of two programs: `rumpsteak-generate` and `subtype`. The former is used to generate a Rust API from a `.dot` file generated by NuScr; the latter is used to check if an automaton (specified in a `.dot` file) is a subtype of an other one.

```
$ git clone git clone --single-branch --branch test_examples \
    https://github.com/zakcutner/rumpsteak.git
$ cd rumpsteak
$ cargo +1.54.0 install --all-features --path .
$ cargo +1.54.0 install --all-features --path ./generate
```

### 3.4. Running examples

Examples are provided in the folder `examples`, in which you can find various `.rs` files, and a `Running_examples` subfolder. The `examples/Running_examples` folder contains itself multiple folders, each of them corresponding to an example.

Go to `examples/Running_examples/3buyers`. The folder contains a file `three_buyers.nuscr`. By running

```
$ nuscr --fsm A@ThreeBuyers three_buyers.nuscr > A.dot
```

we generate the graph corresponding to the process `A` of the protocol. Notice that, due to a limitation of `nuscr`, we have to edit the `A.dot` file to change the name of the graph generated. Therefore, in practice, we run

```
$ nuscr --fsm A@ThreeBuyers three_buyers.nuscr | sed s/G/A/ > A.dot
```

Running a similar command allows to create the graphs for endpoints `C` and `S` too. For the sake of the example, one can compare the obtained files with the expected one (`{A, C, S}_expected.dot`).

With the `.dot` files, we can now generate the Rust API:

```
$ rumpsteak-generate --name ThreeBuyers C.dot S.dot A.dot > 3buyers.rs
```

Again, for the sake of the example, one can compare the resulting `.rs` file with the expected one (`3buyers_expected.rs`).

Finally, the user can write the actual protocol, using the `.rs` file we generated previously. A completed example is shown in `$rumpsteak/examples/3buyers.rs`, and can be run using:

```
$ cargo run --example 3buyers
```

### 3.5. Running the benchmarks

There are two benchmarks, which correspond to Fig. 6 and Fig. 7 in the paper.



**3.5.A. Runtime benchmarks (Fig. 6)** Runtime benchmarks consist of comparing the runtime of a protocol implemented using Rumpsteak vs. several frameworks (namely `ferrite`, `sesh`, `multicrusty`), or with a single-threaded implementation.

**3.5.B. Subtyping benchmarks (Fig. 7)** The subtyping benchmarks are run using the Hyperfine tool.

### 3.6. Analysing the raw data

**3.6.A. Runtime benchmarks** The results used to generate Figure 6 are generated with Criterion. Criterion produces the following tree of files:

```
.
|-- double_buffering
|   |-- 10000
|   |-- 15000
|   |-- 20000
|   |-- 25000
|   |-- 5000
|   |-- ferrite
|   |-- mpstthree
|   |-- report
|   |-- rumpsteak
|   |-- rumpsteak_optimized
|   `-- sesh
|-- fft
|   |-- 1000
|   |-- 2000
|   |-- 3000
|   |-- 4000
|   |-- 5000
|   |-- ferrite
|   |-- mpstthree
|   |-- report
|   |-- rumpsteak
|   |-- rumpsteak_optimized
|   |-- rustfft
|   `-- sesh
|-- report
|   `-- index.html
`-- stream
    |-- 10
    |-- 20
    |-- 30
    |-- 40
```



```
|-- 50
|-- ferrite
|-- mpstthree
|-- report
|-- rumpsteak
|-- rumpsteak_optimized
`-- sesh
```

38 directories, 1 file

**3.6.B. Subtyping benchmarks** Hyperfine generates a CSV file for the results of each benchmark, located at `results/data/*/*.csv`. For each row, this file contains

1. the actual command that was run (`command`);
2. the mean time to execute the command (`mean`);
3. the standard deviation of executing the command (`stddev`);
4. the median time to execute the command (`median`);
5. the mean *user* time to run the command (`user`);
6. the mean *system* time to run the command (`system`);
7. the minimum time to run the command (`min`);
8. the maximum time to run the command (`max`); and
9. the value of the parameter used in the command (`parameter_n`)

where all times are given in seconds. Since we do not care about execution time spent in the kernel (all relevant computation is done in user space) we take only the timings in the `user` column. We provide a script to plot the mean user execution time against the parameter value for each tool.

## 4. Claims supported or not by the artifact

### 4.1. Claims supported by the artifact

- Results presented in Fig. 7 are fairly stable across different machines, even when run in Docker. One should expect to obtain similar plots on their machine.

### 4.2. Claims partially supported by the artifact

- Results presented in Fig. 6 are quite dependent on the number of cores provided by the machine and on other programs being run on the machine simultaneously. The results presented in the paper are run on a 16-core AMD Opteron 6200 Series at 2.6GHz with hyperthreading and 128GB of RAM.

Attempts run on lower-end processors or from within a Docker container may not provide similar results (in particular, the `rustfft` implementation is highly optimised for sequential execution and outperforms approaches



based on message passing on processors with fewer cores). Nonetheless, the performance of Rumpsteak vs. other MPST implementations would remain mostly comparable.

#### 4.3. Claims not supported by the artifact

- NuScr is an external tool not contributed by the authors, and therefore not part of the claims of the paper.

---

## Appendix

Contents of our Dockerfile:

```
#syntax=docker/dockerfile:1
FROM debian:bullseye

RUN apt update && apt install haskell-platform -y && \
    apt install opam ocaml-nox -y && \
    apt install jq pkg-config libssl-dev -y
RUN /usr/bin/cabal update && \
    /usr/bin/cabal install cmdargs \
        ansi-terminal parallel split parsec hashable --lib

WORKDIR /home/

RUN git clone https://bitbucket.org/julien-lange/kmc-cav19.git
RUN cd kmc-cav19 && \
    ghc KMC -threaded && \
    ln -s "$(pwd)/KMC" /usr/local/bin/kmc && \
    cd ..
RUN git clone https://github.com/julien-lange/asynchronous-subtyping.git
RUN cd asynchronous-subtyping/tool && \
    ghc Checker && \
    ln -s "$(pwd)/Checker" /usr/local/bin/concur19 && cd ../../

RUN opam init --disable-sandboxing && \
    eval $(opam env) && opam install nuscr -y
RUN curl https://sh.rustup.rs -sSf > rustup.sh && \
    bash ./rustup.sh --default-toolchain 1.54.0 -y && \
    /root/.cargo/bin/rustup install nightly-2021-07-06

RUN /root/.cargo/bin/cargo install hyperfine
```



```
ENV PATH="/root/.cargo/bin:/root/.opam/default/bin:${PATH}"

RUN ls
RUN git clone --single-branch --branch test_examples \
    https://github.com/zakcutner/rumpsteak.git
WORKDIR /home/rumpsteak/
RUN /root/.cargo/bin/cargo +1.54.0 install --all-features --path .
RUN /root/.cargo/bin/cargo +1.54.0 install --all-features --path ./generate
RUN /root/.cargo/bin/cargo +1.54.0 build --examples --all-features
```