Artifact - Historia: Refuting Callback Reachability with Message-History Logics

ACM Reference Format:

. 2023. Artifact - Historia: Refuting Callback Reachability with Message-History Logics. 1, 1 (July 2023), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

This document explains the artifact for the Historia paper [1]. The goal of this document is to first give a set of instructions for reproducing the experimental results and then give a technical explanation of how the implementation connects to the technical contributions. We have automated the steps to reproduce the results with scripts explained below.

When using Historia on its own, the input is a compiled Android application in the form of an APK file, a CBCFTL specification defined in a JSON format, and an assertion at a location defined by a JSON data structure. Outputs are either "safe" or "alarm" in which case an abstract message history witnessing the alarm will be available.

2 PREREQUISITES - RUNNING THE HISTORIA DOCKER CONTAINER

We have configured the experiments to be run within two Docker containers provided with this artifact. These Docker files may be found in the root directory of this archive and is labeled `historia.docker`. Please follow the instructions to install docker from https://docs.docker.com/engine/install/.

Importing the docker containers can be done with the following command.

```
docker import historia.docker
docker import experimentdb.docker
```

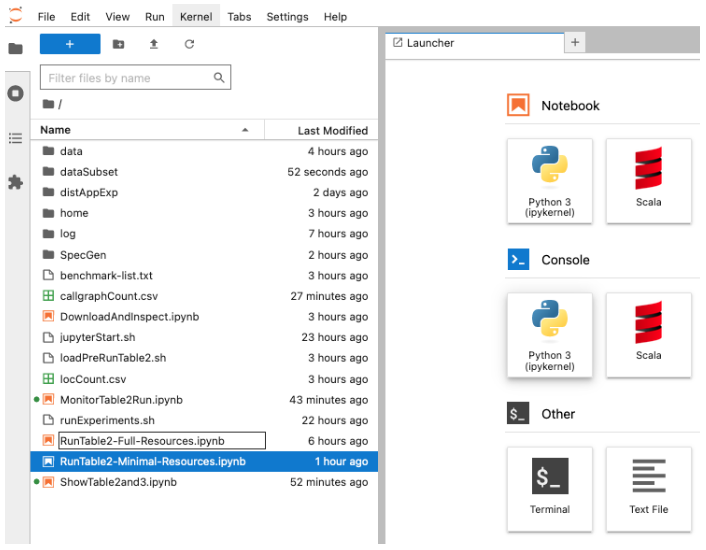
[**TODO:** *double check import works after container creation*]

In order to start the Docker containers, run the following command in the root directory. This will start two Docker containers.

```
docker compose -f docker-compose.yml up
```

All subsequent steps may be done through the web interface at http://localhost:8888. Opening this URL should show a Jupyter notebook as shown by Figure 1.

Author's address:



Name	Last Modified
data	4 hours ago
dataSubset	52 seconds ago
distAppExp	2 days ago
home	3 hours ago
log	7 hours ago
SpecGen	2 hours ago
benchmark-list.txt	3 hours ago
callgraphCount.csv	27 minutes ago
DownloadAndInspect.ipynb	3 hours ago
jupyterStart.sh	23 hours ago
loadPreRunTable2.sh	3 hours ago
locCount.csv	3 hours ago
MonitorTable2Run.ipynb	43 minutes ago
runExperiments.sh	22 hours ago
RunTable2-Full-Resources.ipynb	6 hours ago
RunTable2-Minimal-Resources.ipynb	1 hour ago
ShowTable2and3.ipynb	52 minutes ago

Fig. 1. Your web browser should look like this if the Docker images started correctly.

When finished, the following commands will close down the containers and remove them.

```
docker compose down
docker compose rm
```

System Requirements. We have split the instructions so that a subset of the experiments may be run on a reasonable laptop and included full instructions if access to a server is available. The instructions labeled "minimal resources" may be run on a reasonably modern laptop that has 8GB of ram in addition to ram used by other processes.

Instructions labeled with "full resources" can be run on more powerful systems. We used multiple servers with an AMD EPYC 7763 64-Core Processor and 256GB of ram. Usage of these resources was granted to our project by Chameleon Cloud [**?**].

3 REPRODUCING THE HISTORIA RESULTS

Each subsection here corresponds to a table in the evaluation section. We have labeled the subsections first with the research question (**RQ1** or **RQ2**), the table number (Table 1, Table 2, Table 3), and finally, the resource requirements as described earlier (minimal resources or full resources).

3.1 RQ1 - Table 1 - minimal resources

To run the experiments for the first table, open the terminal in the Jupyter notebook listed earlier and run the following command:

```
./runExperiments.sh
```

The experiments in Table 1 are run as a Scala unit tests defined in the file:
`implementation/src/test/scala/edu/colorado/plv/bounder/symbolicexecutor/Experiments.scala`.

Each benchmark is labeled by `test([description])` where [description] has a row number and short english description. The source code for each benchmark is a string stored in a variable named `src`. This source code is compiled into the APK automatically when running the unit test.

The set of CBCFTL specifications for each row are defined by row in the `ExperimentSpecs` object within `Experiments.scala`. These specifications may be found in `src/main/scala/edu/colorado/plv/bounder/lifestate/Specification.scala`.

The output will be printed to the screen in a log format. Each row of Table 1 may be found labeled by "Row" followed by the row number and version (i.e. "bug" or "fix"). For example, the buggy version of `getAct`[??] is "Row 1 bug" and the fixed version is "Row 1 fix".

Below is an explanation of each column in the table:

(1) Pattern cb,ret - The number of callbacks and returns in the bug pattern: Integer labeled "cbSize".
(2) Pattern ci - the number of callins in the bug pattern: Integer labeled "syntCi".
(3) Historia specs - number of specs written for the benchmark. Integer labeled "spec count". The specific specs may be found in `Experiments.scala` and `Specification.scala` as described earlier.
(4) Historia cb - number of callbacks that may be matched by the CBCFTL spec: Integer labeled "matchedCb".
(5) Historia cbret - number of callback returns that may be matched by the CBCFTL spec: Integer labeled "matchedCbRet".
(6) Historia ci - number of callins matched by the spec: Integer labeled "matchedSyntCi".
(7) Historia time - runtime:
(8) Historia res - The result of the analysis: There will be several rows of text displaying the data from the table. The result of verification labeled with "actual:" and may say "Witnessed" (ⓘ), "Timeout" (🕓), or "Proven" (⊘). For reference, it also prints the expected result after "expected:". If the actual and expected results differ, the unit test fails.

[**TODO:** *SM: explain how to compare flowdroid and infer*]

3.2 RQ2 - Table 2 and Table 3 - Viewing results

Since the full experiments take a long time to run, start by viewing our results in the Jupyter interface. During the experiments, jobs and results are stored in a database in the `historia_postgres` docker container started by `docker compose`. To load the pre-computed results, open a terminal in the Jupyter notebook and run the following script to load the final database:

```
./loadPreRunTable2.sh
```

The results can be viewed by opening the notebook `ShowTable2and3.ipynb` shown on the left. After opening, select "Kernel" -> "Restart Kernel and Run All Cells". This notebook reads the results from the database and displays the Historia results for Table 2 and Table 3 (Shown under heading titles "Table 1" and "Table 2"). Additional statistics about the run may be viewed in this notebook as well. Statistics about the overall runtime may be found toward the end of the notebook `MonitorTable2Run.ipynb`.

Sizes for the call graphs were computed by setting a breakpoint after call graph generation in both Flowdroid and Historia. The call graph was then filtered for methods under the package declared by the application's manifest. We recorded these call graph sizes in the file `callgraphCount.csv`.

3.3 RQ2 - Table 2 - minimal resources

The full set of 1090 locations can take an extremely long time to run (over 15 days of CPU time). Therefore, we recommend running a subset.

The notebook `RunTable2-Minimal-Resources.ipynb` will run by clearing the database and uploading the locations in the "dataSubset" directory. The worker container will then process them one at a time. The directory "dataSubset" contains 10 locations that don't use too much ram and don't time out. Locations are stored as ".json" files. Running should take around 20 minutes depending on the system.

Progress can be tracked through the `MonitorTable2Run.ipynb` notebook. All cells in this notebook may be run repeatedly with "shift-enter" to track the status and outputs. When the `completed_jobs` row under the "Track Total Jobs" heading reaches 10, the subset of locations is completed running.

Once completed, the last cell of the notebook `ShowTable2and3.ipynb` can compare the results just generated against our results. It will print out the path to the config file, the result from the most recent run, and finally, the result from the run used for the paper. Please note that some jobs may not complete depending on the system specs.

Changing the subset. The subset of apps may be changed to a random selection by deleting the contents of the "dataSubset" directory and re-running the last cell of `ShowTable2and3.ipynb`. Please note that many locations will not complete on the 8 GB of ram configured by the docker compose file (see the later instructions to remove this restriction).

3.4 RQ2 - Table 3 - minimal resources

[**TODO:** *notebook and description for table 3 rerun*]

3.5 RQ1 - Table 1 - full resources

In order to run the full version of Table 1, a system with at least 120G of ram must be used. To run, restart the Docker container using the following command:

```
docker compose -f docker-compose-full-resources.yml up
```

Then, repeat the instructions from the "minimal resources" section. This should take about an hour to run on a comparable system.

If this completes successfully, the output should be the same as earlier but will include "Row 4 fix".

3.6 RQ2 - Table 2 - full resources

In order to run the full version of Table 2, we strongly suggest using multiple servers. You can use the `MonitorTable2Run.ipynb` notebook to estimate the expected remaining runtime after a few jobs have completed (See the "Estimate Time Until Completion of All Jobs" cell).

The Docker compose file `docker-compose-full-resources.yml` shows how to start and connect worker containers. Scaling to multiple systems can be done with Docker swarm https://docs.docker.com/engine/swarm/ or simple SSH tunnels connecting port 5432 from the workers to the postgres database container. Once the workers and server are configured, the full experiments may be run through the `RunTable2-Full-Resources.ipynb` notebook. The results may be viewed the same as was explained in the "minimal resources" section earlier.

3.7 RQ2 - Table 3 - minimal resources

4 RUNNING AND INTERPRETING HISTORIA ON CUSTOM INPUTS

[**TODO:** *second priority*]

Historia may be run on arbitrary Android applications as long as an APK can be compiled in debug mode. This is usually accomplished using the command `./gradlew assembleDebug` but will vary from application to application. A location and safety property must be chosen ahead of time. However, we recommend only writing CBCFTL specifications as needed.

Running Historia Through Jupyter. The recommended way to run Historia is using a Jupyter notebook. This allows the inputs to the tool to be defined using the Scala data structures for input rather than JSON.

[**TODO:** *give sample app to walk through this process*]
[**TODO:** *explain this process completely*]
[**TODO:** *explain counter examples*]

5 DEVELOPING FOR HISTORIA

[**TODO:** *fourth priority*]

In this section, we explain the implementation of each technical contribution in Historia.

5.1 Running Unit Tests

[**TODO:** *this may need to go in later connecting formalism section*]

5.2 Application-Only Control-Flow Graph

[**TODO:**]

5.3 Message-History Program Logic (MHPL)

[**TODO:**]

5.4 Callback Control-Flow Temporal Logic (CBCFTL)

[**TODO:**]

5.5 Combining Abstract Message Histories with Callback Control Flow

[**TODO:**]

5.6 Optimizations

[**TODO:**]

REFERENCES

[1] Shawn Meier, Sergio Mover, Gowtham Kaki, and Bor-Yuh Evan Chang. 2023. Historia: Refuting Callback Reachability with Message-History Logics.. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).*