Toeplitz extraction testscript – user manual

This user manual defines how to operate the test script for the Toeplitz extraction module. You can find all relevant files in this Git repository. The test script was developed to allow for easier testing of our iterations during development, and to grant us an easy way of measuring execution times and correctness of each individual implementation. Its primary purpose is to automate testing, timing measurements and result generation with minimal manual intervention, ensuring reproducibility and accuracy, which are crucial for both evaluation and correctness verification of the implementation. The script achieves this by:

- Automating input data transmission via USB serialports
- Capturing and verifying processed data output from MCUs
- Logging detailed timing information as well as displaying average timings
- Generating entropy reports via `ent` to further (*optionally*) evaluating the correctness of each implementation

This approach minimizes human error, allows for consistent benchmarkjing across different iterations and different output bit sizes.

1. Prerequisites

Before using the script, ensure that:

- `python3` is installed
- Required Python modules (*`pyserial`, `argparse`, `glob`, `subprocess`*) are installed in a virtual environment
- The MCU you are testing is connected to your system via USB
- You know the serial port of the connected MCU on your operating system (*for instance `/dev/ttyACM0`, `COM7`, etc.*)

2. Microcontroller Environment

Any implementation that is to be tested by the script needs to be flashed to the connected MCU. More details about how to flash code to the microcontroller in use can be found in its respective user manuals. However, as a general suggestion, for most MCUs you can use either:

- Arduino CLI (*recommended for Linux*)
- Arduino IDE (*recommended for Windows/MacOS*)

If you choose to use Arduino CLI, you will additionally to add the MCUs URLs to the IDE (*for instance, adding the boards for Teensy and Raspberry Pi Pico 2 was necessary in our case*). For other MCUs than the ones noted, refer to the user manual for the controller in question for the correct URL.

These steps demonstrate the process of configuring the two MCUs used in our project as a reference.

```
arduino-cli config init
arduino-cli config add board_manager.additional_urls \
  https://www.pjrc.com/teensy/package_teensy_index.json, \
  https://github.com/earlephilhower/arduino-pico/releases/
  download/global/package_rp2040_index.json
```

Next, we update the index and install cores.

```
arduino-cli core update-index
arduino-cli core install teensy:avr
arduino-cli core install rp2040:rp2040
```

The MCU is now ready to have firmware flashed to it. For Teensy and Pico 2, you can use the supplied Makefile (*`make upload_teensy`* or *`make upload_pico`*).

3. Script Usage

The test script itself is designed to be simple and intuitive to use. When running the script, the CLI instructions should allow you to select the correct serial port of your connected MCU interactively, regardless of your operating system.

```
python3 tests.py --gen
# Generates new baselines for testing
python3 tests.py --test
# Performs tests on the current iteration and verifies
# the result against the appropriate baseline
python3 tests.py --time
# Performs the tests and produces the average execution
# speed of the implementation
```

For each input size (*e.g. 64 bits, 128 bits, etc.*) a new baseline needs to be generated with the baseline implementation for that particular bitsize. This is required as the output will differ depending on the size of the input and output. Do note that the `test` and `time` flags require a flag to be toggled in the firmware to produce the correct output from the MCU. More details about this can be found in section 5.

4. File Structure

The test script depends on a specific file structure in its subfolders. These folders are populated by the test script, and do not require any manual intervention – however, if issues should arise, these folders may need to be cleaned manually.

- `data/` - Input binary test files (*pre sampled/generated entropy data – more files can be added here for more tests*)
- `results/{output_size}/` - Output binaries and entropy reports based on output size of bits
- `logs/iter{iteration}/{output_size}/` - Timing logs for each test iteration and output size of bits

5. Step-by-Step instructions

This process assumes you have generated a baseline using the `--gen` flag for the bit size and iteration you are testing. For maximum correctness, we recommend using `iter1` for generating the baseline.

1. Connect the MCU you wish to test to the host computer via USB. You may want to verify that the port can be found manually before proceeding.
2. Flash the appropriate firmware to the microcontroller. The test script depends on the data being packed as detailed in the `loop()` of each iteration as shown in the repository. Flash the code to the MCU using either Arduinos CLI or the IDE, depending on your personal preference.

Depending on the type of test you wish to run, you may need to change the value of the globally defined variable `TIMED` at the top of the file – `0` if you wish to test the correctness of the implementation, `1` to test the execution speed. considering that the output from the MCU needs to change depending on if you want to verify the correctness or the execution time, this setting needs to be adjusted on the MCU before running the script. Furthermore, you may need to change the variable `RAW_BITS_LEN` to match the length of the input bits you are testing.

3. Executing the script is now as simple as running the appropriate command as listed in section 3, depending on whether you want to test your implementations correctness or measure the execution time.

If all previous steps are done correctly, the test script will execute and provide appropriate feedback on the command line depending on the selected modes.

Division of labor

During this project, our division of labor came quite naturally. To begin with, it is worth noting that we worked side-by-side roughly 60% of our effective time spent on the project, and the remaining 40% of time was spent working individually on parts of the project. During the periods of time where we worked side-by-side, we divided the work equally amongst each other depending on what was currently needed. One specific example worth mentioning is our programming work – for instance, there was a slight division of labor here in terms of the written code. For instance, Nadim may have written and commited more code related to the microcontrollers, whereas Love may have written and commited more code related to the test script – but these two seaprate codebases are intrinsically linked. The test script cannot function without an intimate knowledge of the microcontroller code, and the opposite is also true to a certain extent.

The periods of time where we worked separately, we did not have an explicit division of labor – however, we had during our discussions working side by side ascertained a loose division of labor of what was left to be done from our previous sessions. We did not encounter any periods of time where we were behind our planned schedule, instead finding more time for writing more iterations and including them in the thesis.

In terms of writing the main thesis text, this was also a primarily collaborative effort. Here, we divided certain parts of the text between each other, but afterward we reviewed the parts written by the other and discussed any potential issues that may have been found. There were certain parts that we divided between each other – one concrete example is that Love focused more on sections 3.1 and 3.2, whereas Nadim focused more on sections 3.3 and 3.4 in the background section. This division was not discussed at great length, instead, we mainly talked about which part interested us the most and worked on these.

This quite loose structure of labor division worked surprisingly well for our project, and neither of us has encountered any issues in how the work was divided. Considering that the project was in large parts was quite theoretical and required more thought and discussion rather than produced code, this worked to our advantage. However, it is worth noting that a project in which more code would need to be produced, it is very likely that this method of work may not be as successful. We had the opportunity to work more on quality of each individual piece of the proverbial puzzle rather than having an abundance of work to do, which worked excellently for our particular case.

It is worth noting that we did encounter some issues during our work, however, these were not connected to our division of labor. Our initial experiments, as we planned them out, proved to ultimately be fruitless as none of our initial implementations provided any tangible benefits to the execution time of the algorithm. This meant that we had to have many discussions about how to move forward, and this led to a redevelopment of our methodology – adding in more iterations to the thesis as we tested them in practice. In hindsight, this issue arose due to the structure of the previous course in conjunction with the actual thesis work, in which we needed to formulate a tangible methodology out of a theoretical concept before we had started the actual work. It is worth noting that we had worked on a project together before this one, which meant we had a good pre-existing working relationship going into the thesis. Without this previous experience, this looser structure we used may very well have been detrimental to our work.

In summary, our division of labor came very naturally during the course of the project. Considering the comparatively limited scope of the thesis, this allowed us to focus more time on each individaul part of the project, and discuss the findings and issues as well as coming up with solutions together. Openness and transparency, as well as personal responsibility has been the foundational cornerstones of our collaborative effort, which has been quite excellent from the start.