Let us walk on the 3-isogeny graph: Step-by-step Artifact Walkthrough

Jesús-Javier Chi-Domínguez, Eduardo Ochoa-Jiménez and Ricardo-Neftalí Pontaza-Rodas

August 17, 2025

Outline

Repository Overview (1/4)

- *Let us walk on the 3-isogeny graph: efficient, fast, and simple* is an open-source C framework for using 3-radical isogenies to improve some post-quantum cryptosystems (dCTIDH + QFESTA).
- This presentation summarizes the software structure and reproducibility workflow.

Repository Overview (2/4)

- Hosted on GitHub: `https://github.com/Crypto-TII/pqc-engineering-ssec-23`
- Modular design with components: Presentation Video, System Requirements, Build, Test, Benchmarks, Docs, Manuscript results replication, and CI/CD Pipeline.

Repository Overview (3/4)

Overview of our paper - YouTube video:
https://www.youtube.com/watch?v=BjedMooSV30&list=PLFgwYy6Y-xWYCFruq66CFXXiWEWckEk6Q

Figure 1: Overview of our paper - YouTube video.

Repository Overview (4/4)

(Full) Guided Tour of our Artifact:
https://www.youtube.com/watch?v=hLk_B5NpKRA&list=PLFgwYy6Y-xWYCFruq66CFXXiWEWckEk6Q&index=10

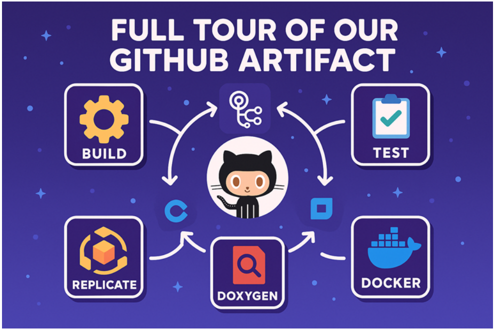


Figure 2: (Full) Guided Tour - YouTube video.

System Requirements

Our system requirements are extremely simple:

1. Out-of-the-box Linux (CPU Intel x86_64).
2. CMake + gcc
3. Python3:
   - Numpy
   - Matplotlib

How to Install?

Clone from GitHub.
Run:

```
git clone
https://github.com/Crypto-TII/pqc-engineering-ssec-23.git
```

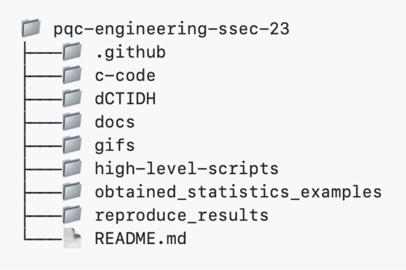


Figure 3: Downloaded project structure.

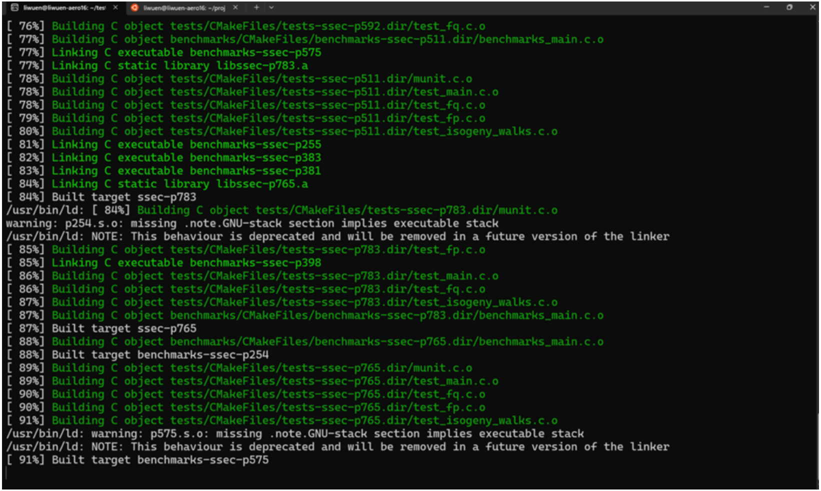
How to Build? (1/2)

Run:

```
cd c-code
cmake -DCMAKE_BUILD_TYPE=Release -B cmake-build-release
cd cmake-build-release
make -j
```

Figure 4: Build instructions.

How to Build? (2/2)



```
[ 76%] Building C object tests/CMakeFiles/tests-ssec-p592.dir/test_fq.c.o
[ 77%] Building C object benchmarks/CMakeFiles/benchmarks-ssec-p511.dir/benchmarks_main.c.o
[ 77%] Linking C executable benchmarks-ssec-p575
[ 77%] Linking C static library libssec-p783.a
[ 78%] Building C object tests/CMakeFiles/tests-ssec-p511.dir/munit.c.o
[ 78%] Building C object tests/CMakeFiles/tests-ssec-p511.dir/test_main.c.o
[ 78%] Building C object tests/CMakeFiles/tests-ssec-p511.dir/test_fq.c.o
[ 79%] Building C object tests/CMakeFiles/tests-ssec-p511.dir/test_fp.c.o
[ 80%] Building C object tests/CMakeFiles/tests-ssec-p511.dir/test_isogeny_walks.c.o
[ 81%] Linking C executable benchmarks-ssec-p255
[ 82%] Linking C executable benchmarks-ssec-p383
[ 83%] Linking C executable benchmarks-ssec-p381
[ 84%] Linking C static library libssec-p765.a
[ 84%] Built target ssec-p783
/usr/bin/ld: [ 84%] Building C object tests/CMakeFiles/tests-ssec-p783.dir/munit.c.o
warning: p254.s.o: missing .note.GNU-stack section implies executable stack
/usr/bin/ld: NOTE: This behaviour is deprecated and will be removed in a future version of the linker
[ 85%] Building C object tests/CMakeFiles/tests-ssec-p783.dir/test_fp.c.o
[ 85%] Linking C executable benchmarks-ssec-p398
[ 86%] Building C object tests/CMakeFiles/tests-ssec-p783.dir/test_main.c.o
[ 86%] Building C object tests/CMakeFiles/tests-ssec-p783.dir/test_fq.c.o
[ 87%] Building C object tests/CMakeFiles/tests-ssec-p783.dir/test_isogeny_walks.c.o
[ 87%] Building C object benchmarks/CMakeFiles/benchmarks-ssec-p783.dir/benchmarks_main.c.o
[ 87%] Built target ssec-p765
[ 88%] Building C object benchmarks/CMakeFiles/benchmarks-ssec-p765.dir/benchmarks_main.c.o
[ 88%] Built target benchmarks-ssec-p254
[ 89%] Building C object tests/CMakeFiles/tests-ssec-p765.dir/munit.c.o
[ 89%] Building C object tests/CMakeFiles/tests-ssec-p765.dir/test_main.c.o
[ 90%] Building C object tests/CMakeFiles/tests-ssec-p765.dir/test_fq.c.o
[ 90%] Building C object tests/CMakeFiles/tests-ssec-p765.dir/test_fp.c.o
[ 91%] Building C object tests/CMakeFiles/tests-ssec-p765.dir/test_isogeny_walks.c.o
/usr/bin/ld: warning: p575.s.o: missing .note.GNU-stack section implies executable stack
/usr/bin/ld: NOTE: This behaviour is deprecated and will be removed in a future version of the linker
[ 91%] Built target benchmarks-ssec-p575
```

Figure 5: Build process demo.

How to Run Tests? (1/2)

For running unit tests, simply execute:

- `cd cmake-build-release`
- `./tests/tests-ssec-p254`
- `./tests/tests-ssec-p255`
- `./tests/tests-ssec-p381`
- `./tests/tests-ssec-p383`
- `./tests/tests-ssec-p398`
- `./tests/tests-ssec-p511`
- `./tests/tests-ssec-p575`
- `./tests/tests-ssec-p592`
- `./tests/tests-ssec-p765`
- `./tests/tests-ssec-p783`

How to Run Tests? (2/2)

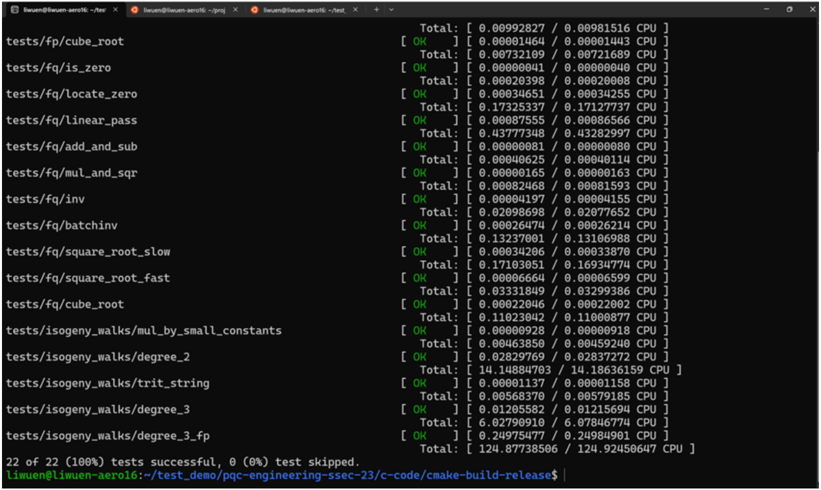


Figure 6: Test demo.

How to Benchmark? (1/4)

- **Important:** Need to use flags `-DCMAKE_BUILD_TYPE=Release` `-DBENCHMARKING=CYCLES` when building.
- If the flags were not used, the benchmarks will be empty.

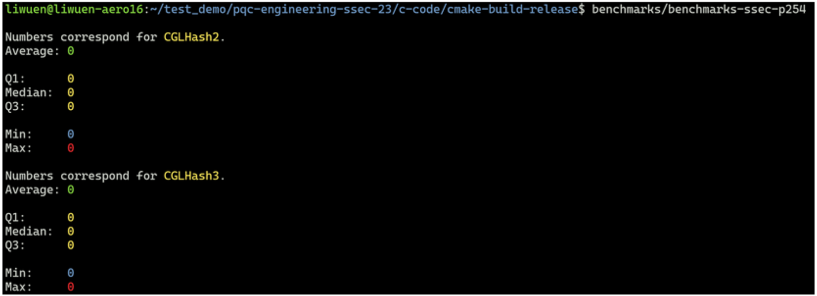


Figure 7: Benchmark errors

How to Benchmark? (2/4)

To **build** benchmarking, simply execute:

- `cmake -DCMAKE_BUILD_TYPE=Release -DBENCHMARKING=CYCLES -DARCHITECTURE=x8664 -B cmake-build-release-cycles-x8664`
- `cd cmake-build-release-cycles-x8664`
- `make -j`

How to Benchmark? (3/4)

To **run** benchmarking, inside the `cmake-build-release-cycles-x8664` folder, simply execute:

- `benchmarks/benchmarks-ssec-p254`
- `benchmarks/benchmarks-ssec-p255`
- `benchmarks/benchmarks-ssec-p381`
- `benchmarks/benchmarks-ssec-p383`
- `benchmarks/benchmarks-ssec-p398`
- `benchmarks/benchmarks-ssec-p511`
- `benchmarks/benchmarks-ssec-p575`
- `benchmarks/benchmarks-ssec-p592`
- `benchmarks/benchmarks-ssec-p765`
- `benchmarks/benchmarks-ssec-p783`

How to Benchmark? (4/4)

```
liwuen@liwuen-aero16:~/test_demo/pqc-engineering-ssec-23/c-code/cmake-build-debug-cycles-x8664$ benchmarks/
CMakeFiles/            benchmarks-ssec-p381  benchmarks-ssec-p511  benchmarks-ssec-p765
benchmarks-ssec-p254   benchmarks-ssec-p383  benchmarks-ssec-p575  benchmarks-ssec-p783
benchmarks-ssec-p255   benchmarks-ssec-p398  benchmarks-ssec-p592
liwuen@liwuen-aero16:~/test_demo/pqc-engineering-ssec-23/c-code/cmake-build-debug-cycles-x8664$ benchmarks/benchmarks-ssec
-p254

Numbers correspond for CGLHash2.
Average: 155026965

Numbers correspond for CGLHash3.
Average: 152084312

liwuen@liwuen-aero16:~/test_demo/pqc-engineering-ssec-23/c-code/cmake-build-debug-cycles-x8664$ benchmarks/benchmarks-ssec
-p255

Numbers correspond for CGLHash2.
Average: 155490448

Numbers correspond for CGLHash3.
Average: 134325940

liwuen@liwuen-aero16:~/test_demo/pqc-engineering-ssec-23/c-code/cmake-build-debug-cycles-x8664$ benchmarks/benchmarks-ssec
-p381

Numbers correspond for CGLHash2.
Average: 353393318

Numbers correspond for CGLHash3.
Average: 375604278

liwuen@liwuen-aero16:~/test_demo/pqc-engineering-ssec-23/c-code/cmake-build-debug-cycles-x8664$
```

Figure 8: Benchmarking Demo

Reproducing Manuscript's Graphics

- Scripts located in `reproduce_results` folder.
- Need Python: Numpy and Matplotlib.

```
📁 pqc-engineering-ssec-23
├──📁 c-code
├──📁 dCTIDH
├──📁 docs
├──📁 gifs
├──📁 high-level-scripts
├──📁 obtained_statistics_examples
├──📁 reproduce_results
│   ├──📁 manuscript_figure_03
│   │   ├──📄 benchmark_graph_03.py
│   │   └──📄 generate_figure_03.sh       # <= NEED TO EXECUTE
│   ├──📁 manuscript_figure_04
│   │   ├──📄 benchmark_graph_04.py
│   │   └──📄 generate_figure_04.sh       # <= NEED TO EXECUTE
│   └──📁 manuscript_figure_05
│       ├──📁 dCTIDH_benchmarks_output    # <= AUTOMATICALLY GENERATED!
│       ├──📁 dCTIDH_builds               # <= AUTOMATICALLY GENERATED!
│       ├──📁 statistics_output           # <= AUTOMATICALLY GENERATED!
│       ├──📄 analyze_bench.py
│       ├──📄 benchmark_graph_05.py
│       └──📄 generate_figure_05.sh       # <= NEED TO EXECUTE
└──📄 README.md
```

Figure 9: Location of bash scripts to reproduce manuscript's results.

Reproducing Manuscript's Graphics: Figure 3 (1/4)

Simply execute

- `cd reproduce_results/manuscript_figure_03`
- `chmod +x generate_figure_03.sh`
- `./generate_figure_03.sh`

Reproducing Manuscript's Graphics: Figure 3 (2/4)

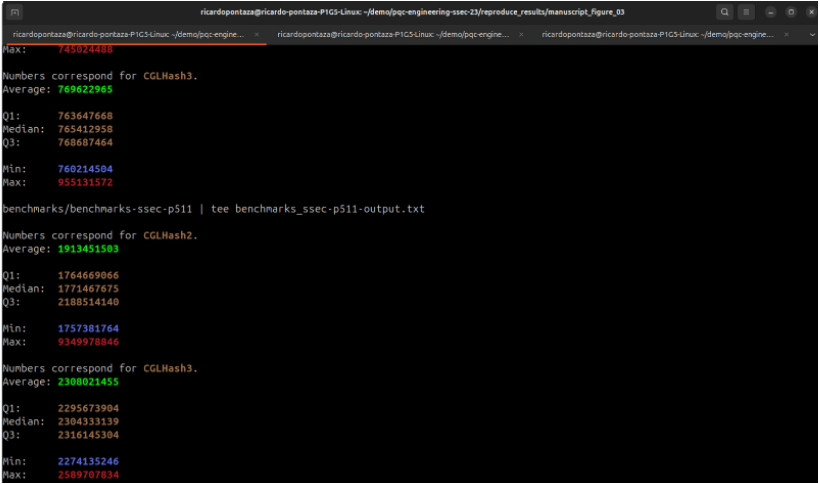


Figure 10: Generation script for Figure 3.

Reproducing Manuscript's Graphics: Figure 3 (3/4)

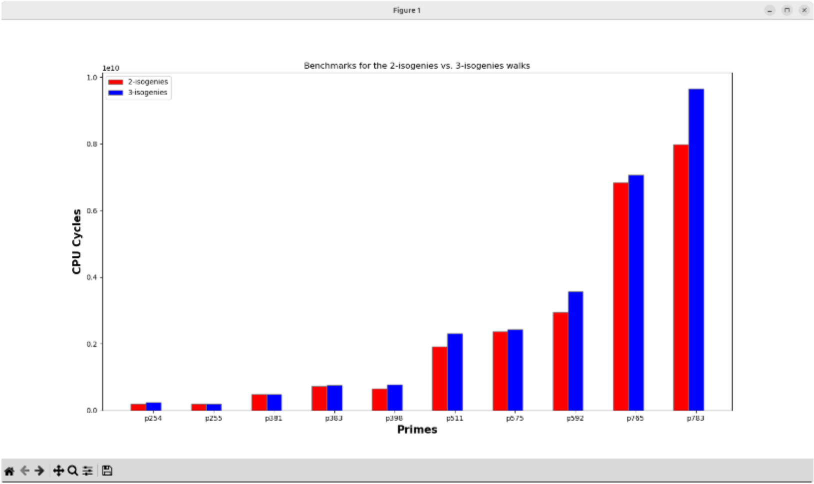


Figure 11: Generated statistical results from `generate_figure_03.sh`

Reproducing Manuscript's Graphics: Figure 3 (4/4)

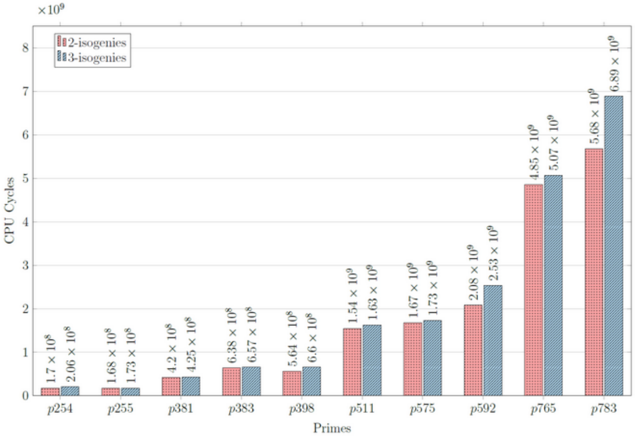


Figure 3: Benchmarks for the 2-isogenies vs. 3-isogenies walks, measured in CPU cycles.

Figure 12: Manuscript's Figure 3.

Reproducing Manuscript's Graphics: Figure 4 (1/4)

Simply execute

- `cd reproduce_results/manuscript_figure_04`
- `chmod +x generate_figure_04.sh`
- `./generate_figure_04.sh`

Reproducing Manuscript's Graphics: Figure 4 (2/4)

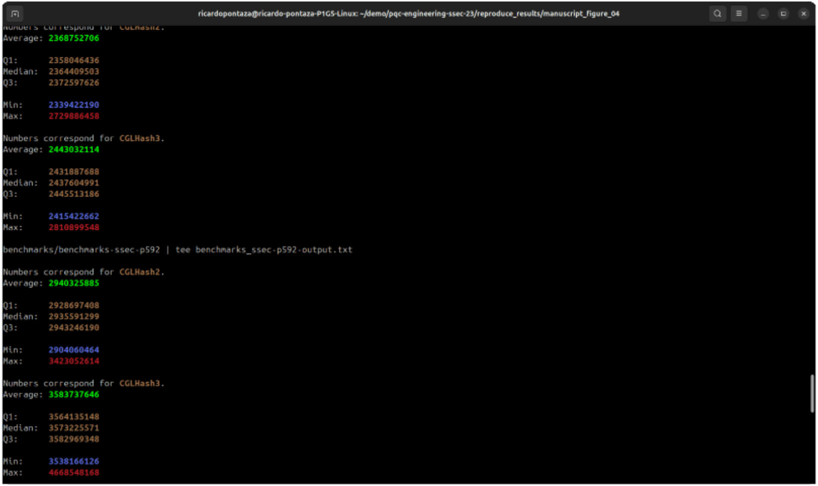


Figure 13: Generation script for Figure 4.

Reproducing Manuscript's Graphics: Figure 4 (3/4)

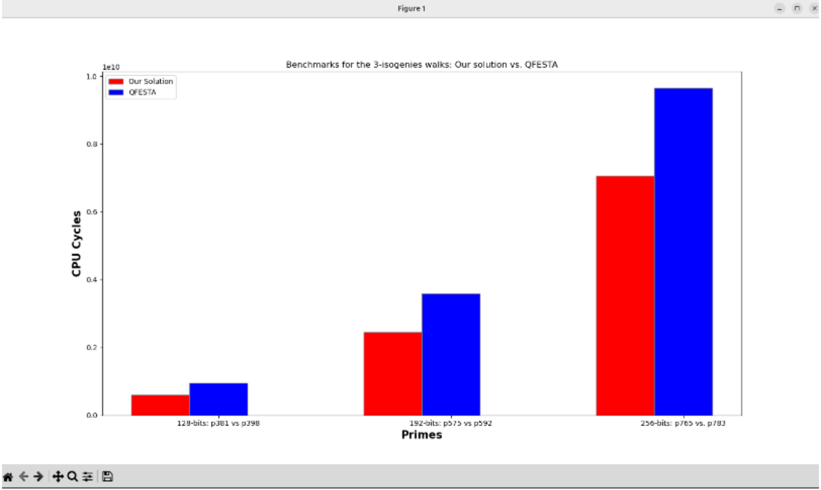


Figure 14: Generated statistical results from `generate_figure_04.sh`

Reproducing Manuscript's Graphics: Figure 4 (4/4)

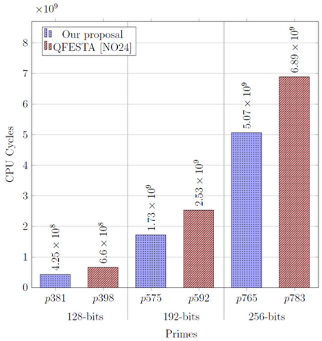


Figure 4: Benchmarks for the 3-isogenies walks for our proposed primes ($p381$, $p575$ and $p765$) vs. QFESTA [NO24] primes ($p398$, $p592$ and $p783$). Both $p381$ and $p398$ offer 128-bits security, while $p575$ and $p592$ offer 192-bits security, and $p765$ and $p783$ offer 256-bits security. For these six primes, the performance was measured in CPU cycles, having an improvement of 35,60% for 128-bits, 31.62% for 192-bits, and 26.41% for 256-bits, respectively.

Figure 15: Manuscript's Figure 4.

Reproducing Manuscript's Graphics: Figure 5 (1/5)

Simply execute

- `cd reproduce_results/manuscript_figure_05`
- `chmod +x generate_figure_05.sh`
- `./generate_figure_05.sh`

Reproducing Manuscript's Graphics: Figure 5 (2/5)

- The previous commands will (automatically) generate some folders.
- You can delete these (automatically-generated) folders between each run if necessary.

```
📁 manuscript_figure_05
├──📁 dCTIDH_benchmarks_output    # <= AUTOMATICALLY GENERATED!
├──📁 dCTIDH_builds               # <= AUTOMATICALLY GENERATED!
├──📁 statistics_output           # <= AUTOMATICALLY GENERATED!
├──📄 analyze_bench.py
├──📄 benchmark_graph_05.py
└──📄 generate_figure_05.sh       # <= NEED TO EXECUTE
```

Figure 16: Automatically-generated folders.

Reproducing Manuscript's Graphics: Figure 5 (3/5)

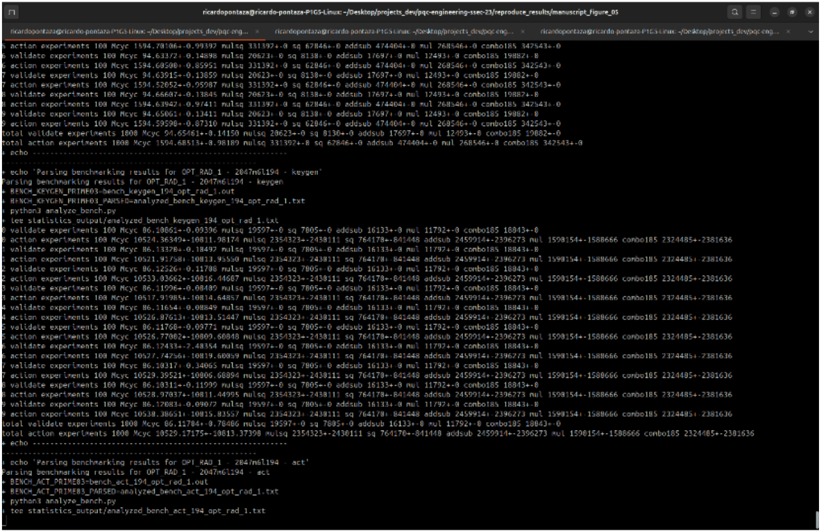


Figure 17: Generation script for Figure 5.

Reproducing Manuscript's Graphics: Figure 5 (4/5)

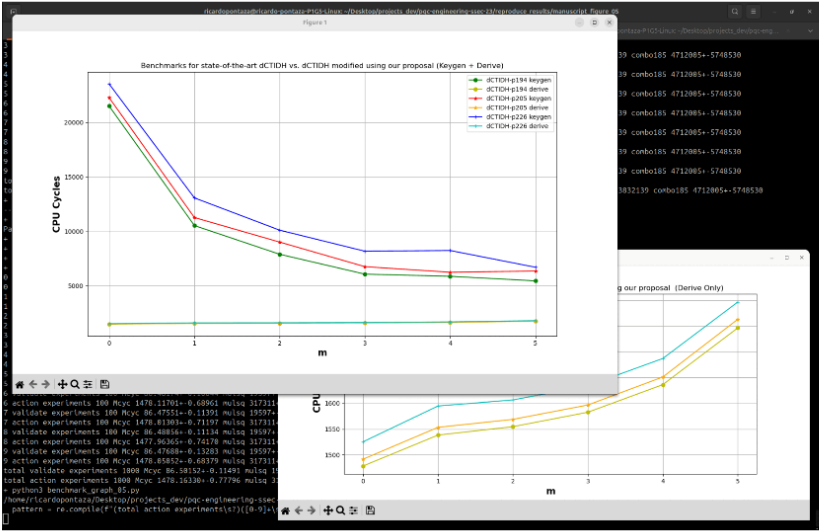


Figure 18: Generated statistical results from `generate_figure_05.sh`

Reproducing Manuscript's Graphics: Figure 5 (5/5)

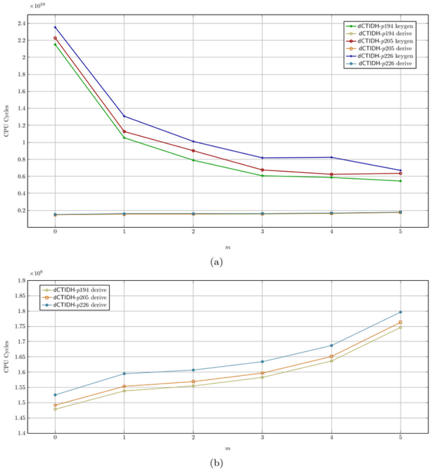


Figure 5: Benchmarks for state-of-the-art dCTIDH vs. dCTIDH modified using our proposal. Both the key generation (keygen) and the shared key derivation (derive) were tested. From

Figure 19: Manuscript's Figure 5.

Generating Technical Documentation (1/2)

We use Doxygen to generate the technical documentation.

- Configuration file: `Doxyfile`
- To generate, simply execute:
  - `cd docs`
  - `doxygen Doxyfile`
- Output in `docs/html/index.html`

Public link:

`https://crypto-tii.github.io/pqc-engineering-ssec-23/`

Generating Technical Documentation (2/2)

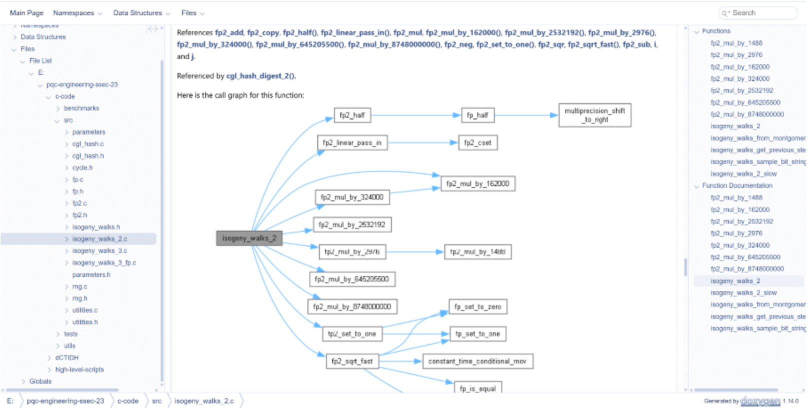


Figure 20: Technical documentation generated using Doxygen.

CI/CD Overview (1/3)

In order to show that our project can be integrated in a Real-World industrial environment, we provide a CI/CD pipeline.

- We use GitHub Actions for CI/CD.
- Pipeline includes **Build**, **Test**, **Benchmark**, and **Reporting** stages.
- YAML config: `.github/workflows/cmake-multi-platform.yml`

CI/CD Overview (2/3)

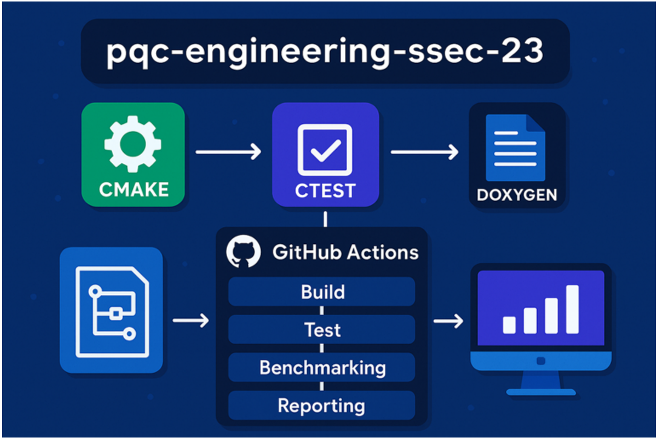


Figure 21: Designed CI/CD pipeline with **Build**, **Test**, **Benchmarking**, and **Reporting** stages.

CI/CD Overview (3/3)

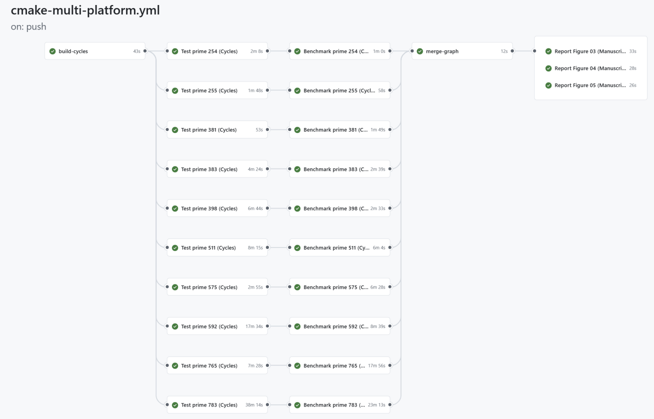


Figure 22: Pipeline in action, running with GitHub actions.

CI Stage: Build

- Triggers on push and pull request
- Any linux (Ubuntu example), Intel x86_64 CPU.
- Uses CMake caching for speed.

CI Stage: Test

- Runs unit and integration tests
- Stores artifacts for future analysis
- Automatic failure reports

CI Stage: Benchmark

- Executes performance benchmarks in both **CPU cycles** and **execution nanoseconds**.
- Benchmarking for every proposed prime.

CI Stage: Reporting (1/2)

- All the scripts used to reproduce our results reported in the manuscript are tested.
- The generated statistical data and the generated graphs are uploaded as public artifacts in our GitHub pipeline so they can be used freely.
- This allows collaborators, scientists, and anyone in general to reproduce, validate, and expand our research project.

CI Stage: Reporting (2/2)

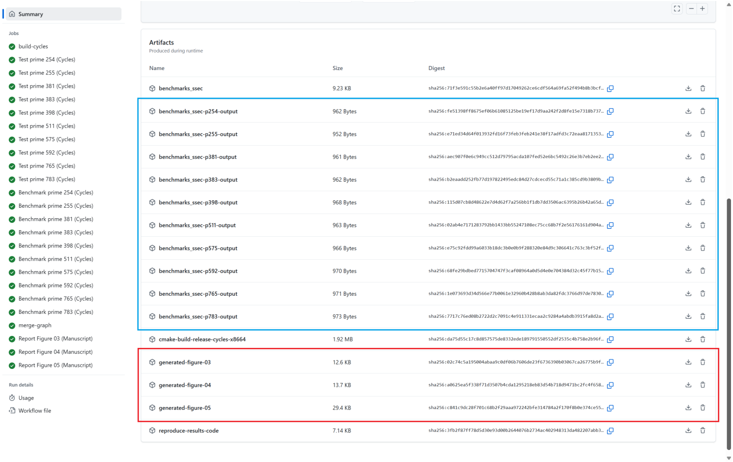


Figure 23: Publicly available artifacts.

Docker Container

Simply execute

- `docker pull tiicrc/github-selfhosted-runner-pqc:latest`
- `docker images | grep pqc`

Docker Container (2/2)

To mount, first locate your terminal at the artifact's root folder (*pqc-engineering-ssec-23*) and execute

- `docker run --rm -ti -v $PWD:/src -w /src tiicrc/github-selfhosted-runner-pqc:latest bash`

After mounting, the terminal will change to

- `/src# <insert commands here>`

Industrial Readiness Proof-of-Concept

- Simulates real deployment environments
- Documented logs, errors, and benchmarking outputs

Additional Resources: CPU benchmarking

- Included details on how to:
  - Turn off turbo-boost.
  - Assembly instructions used in our benchmarking.
- Automated benchmarking scripts under `high-level-scripts/benchmark_02_20250408.sh`

License and Contributions

- Open-source under Apache License.
- License guidelines in `LICENSE` file.
- Issues and PRs welcome!

About the Authors

- Jesús-Javier Chi-Domínguez,
- Eduardo Ochoa-Jiménez,
- Ricardo-Neftalí Pontaza-Rodas.

Thank you


LET US WALK ON
THE 3-ISOGENY GRAPH:
EFFICIENT, FAST, AND SIMPLE
POST-
QUANTUM
THANK YOU

Thank you

Thank you!