USENIX Security '25 Artifact Appendix: "SoK: Understanding zk-SNARKs: The Gap Between Research and Practice"

Junkai Liang[1,*], Daqi Hu[1,*], Pengfei Wu[2,*], Yunbo Yang[3], Qingni Shen[1,†], Zhonghai Wu[1,†]
[1]Peking University, [2]Singapore Management University, [3]East China Normal University
{ljknjupku, hudaqi0507}@gmail.com, pfwu@smu.edu.sg, yyb9882@gmail.com,
{qingnishen, wuzh}@pku.edu.cn

A Artifact Appendix

A.1 Abstract

As the inherent mathematical complexity has hindered the development of zk-SNARK fields, our artifact helps bridge the gaps between researchers, developers and users. The artifact consists of three main aspects: 1)sample code. We provide each sample code with a Docker image that allows users to conduct preliminary testing of these zkSNARK libraries. 2)tutorial. A detailed tutorial explaining the logical construction in the sample programs. 3)extended documentation (wiki book). This document outlines the circuit construction for each library's frontend, the zk-SNARK proving process in the backend, relevant APIs for library gadgets, and supported elliptic curves.

A.2 Description & Requirements

Our artifact contains several Docker virtual images that help practitioners test the efficiency of the three sample zk-SNARKs applications. The images can be run on any personal computer without installing the library.

A.2.1 Security, privacy, and ethical concerns

All evaluated zk-SNARK libraries are open-source and freely available on GitHub or their respective homepages. As such, this research does not involve any ethical concerns, as it does not include activities that could pose harm or risk to individuals or organizations.

A.2.2 How to access

Stable URL pointing to the latest version of our artifact: https://doi.org/10.5281/zenodo.14682405.

Stable URL for the feedback and dynamic changes: https://github.com/zk-lover/zk-sok.

A.2.3 Hardware dependencies

The codes in our artifact can be run on any general-purpose computer. The evaluation requires no special hardware features.

A.2.4 Software dependencies

We test our code on Ubuntu-20.04 with docker-26.1.4. The evaluation does not require specific OS or other software dependencies.

A.2.5 Benchmarks

None.

A.3 Set-up

We have packaged all the environments required for the experiments into Docker, so you don't need to prepare any environment except installing Docker.

A.3.1 Installation

Run the following instructions on any terminal in Linux operating system.

1. sudo apt update
2. sudo apt upgrade
3. apt-get install ca-certificates curl gnupg lsb-release
4. curl -fsSL http://mirrors.aliyun.com /docker-ce/linux/ubuntu/gpg
5. sudo apt-key add -
6. sudo add-apt-repository "deb [arch=amd64] http://mirrors.aliyun.com/docker-ce/linux /ubuntu $(lsb_release -cs) stable"
7. apt-get install docker-ce docker-ce-cli containerd.io

A.3.2 Basic Test

Taking Arkworks as an example, you can enter the /arkworks-lab directory and execute the following command to start Docker.

```
docker build -t arkworks .
docker run -it –rm arkworks
```

Then you can run our sample programs. For example:

```
cargo run –bin cubic_expression
```

A successful run will display output similar to the following:

```
Number of constraints: 3
Uncompressed pk size: 3984 bytes
Uncompressed vk size: 872 bytes
Total uncompressed size (pk + vk): 4856 bytes
Number of constraints: 3
Uncompressed proof size: 384 bytes
Proof is valid: true
Prove time: 74.518 milliseconds
Verify time: 51.528 milliseconds
```

You can use the following command to test the other two sample programs.

```
cargo run –bin rangeproof
cargo run –bin sha256
```

You can easily test the other eight libraries we evaluated using similar commands. We have packaged the code and environment into Docker, and you will see similar output.

A.4 Evaluation workflow

A.4.1 Major Claims

The main claim in our paper related to the artifact is in Section 5.4.3, and we explain the relation in detail as follows:

Our zk-sok open source repository evaluated a total of 9 libraries and implemented our three sample programs using the ZKP schemes they support, Cubic expression, Range proof and sha256. For different application scenarios, we recommend the best scheme along with its implementation. Groth16 is the best practice for applications that need a fast prover, a small proof size, and can tolerate a trust setup. gnark implements Groth16 more efficiently in Go, while snarkjs provides an implementation of Groth16 in Rust with more compatibility (using a DSL compiler). Plonk is the best practice for applications that need a transparent setup and are not sensitive to the slight increase in the proof size. For the widely used range proof, we recommend dalek, which is designed for range proof, specifically. This is proven by the experiment (E1-E9) described in [5.4 Experimental Evaluation] whose results are illustrated/reported in [Table 4].

A.4.2 Experiments

Our evaluation consists of 9 evaluations of the libraries. The instructions and descriptions are listed as follows:

(E1): *[gnark] [30 human-minutes + 5 compute-minutes +1GB disk]:*

Preparation: *No need to configure any environment except Docker*

Execution: Navigate to the /gnark directory:

```
cd /gnark
docker build -t gnark .
docker run -it –rm gnark
```

Navigate to the directory of the program you would like to run. Our examples are in the directories under /root. Run the following code to execute the examples.

```
cd testCubicequation_groth16
go build -o testCubicequation_groth16
./testCubicequation_groth16
```

(E2): *[libsnark] [30 human-minutes + 5 compute-minutes]:*

Preparation: Before building docker, execute:

```
git submodule update –init –recursive libsnark-lab/deps/libsnark
```

Execution: Navigate to the /libsnarklab directory:

```
cd /libsnarklab
Run docker:
docker build -t libsnark .
docker run -it –rm libsnark
```

Navigate to the directory of the program you would like to run. Our examples are at /app/src. Before you execute the sample program, you need run the following code to compile it.

```
mkdir build
cd build
cmake ..
make
```

Then you can run the following code to execute the sample program.

```
cd app/build/src
./cubic_expression
./range_proof
./sha256
```

(E3): *[libiop] [30 human-minutes + 5 compute-minutes]:*
Preparation: Before building docker, execute:

```
git submodule update –init –recursive libioplab/deps/libiop
```

Execution: Navigate to the /libioplab directory:

```
cd /libioplab
Run docker:
docker build -t libiop .
docker run -it –rm libiop
```

Navigate to the directory of the program you would like to run. Our examples are at /app/src. Before you execute the sample program, you need to run the following code to compile it.

```
mkdir build
cd build
cmake ..
make
```

Then you can run the following code to execute the sample program.

```
cd app/build/src
./cubic_aurora
./cubic_ligero
./cubic_fractal
./rangeproof_aurora
./rangeproof_ligero
./rangeproof_fractal
./sha256_aurora
./sha256_ligero
./sha256_fractal
```

(E4): *[snarkjs] [30 human-minutes + 5 compute-minutes + 10 compilation-hours]:*
Preparation: No need to configure any environment except Docker

Execution: Navigate to the /snarkjs directory:

```
cd /snarkjs
Run docker:
docker build -t snarkjs .
docker run -it –rm snarkjs
```

Navigate to the directory of the program you would like to run. Our examples are at /app. Enter a directory, you'll find 4 files: a .circom file, which is the circuit file and input.json file, which is the input file a start.sh file, which contains the full steps to run the program. a quick_start.sh file, which can be used when you have a final.ptau file. The final.ptau file contains the public parameters required by the circuit and is used to ensure the credibility of the circuit. It is a necessary file in the proof process. Snarkjs provides a command to generate this file. As the maximum constraints of the circuit increase, the file generation time is slower. Snarkjs officially provides final.ptau files ranging from 256 constraints to 256 mega constraints. You can choose to generate it yourself or download it. tips: To test sha256 requires a big ptau file, if you generate it yourself, it will take a long time. You can simply run the 'start.sh' to execute the examples.

```
./start.sh
```

Or when you have a final.ptau file, you can run the 'quick_start.sh' to execute the examples.

(E5): *[halo2] [30 human-minutes + 5 compute-minutes]:*
Preparation: No need to configure any environment except Docker
Execution: Navigate to the /halo2 directory:

```
cd /halo2
Run docker:
docker build -t halo2 .
docker run -it –rm halo2
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run –bin cubic_expression
cargo run –bin range_proof
cargo run –bin sha256
```

(E6): *[delak] [30 human-minutes + 5 compute-minutes]:*
Preparation: No need to configure any environment except Docker.
Execution: Navigate to the /delak directory:

```
cd /delak
Run docker:
docker build -t delak .
docker run -it –rm delak
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run –bin rangeproof
```

(E7): *[arkworks] [30 human-minutes + 5 compute-minutes]:*
Preparation: No need to configure any environment except Docker

Execution: Navigate to the /arkworkslab directory:

```
cd /arkworkslab
Run docker:
docker build -t arkworks .
docker run -it –rm arkworks
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run –bin cubic_expression
cargo run –bin rangeproof
cargo run –bin sha256
```

(E8): *[spartan] [30 human-minutes + 5 compute-minutes]:*
Preparation: No need to configure any environment except Docker.
Execution: Navigate to the /spartan directory:

```
cd /spartan
Run docker:
docker build -t spartan .
docker run -it –rm spartans
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run –bin cubic_expression
cargo run –bin rangeproof
cargo run –bin sha256
```

(E9): *[plonky2] [30 human-minutes + 5 compute-minutes]:*
Preparation: No need to configure any environment except Docker.
Execution: If you want to run Cubic expression and Range proofs, navigate to the /plonky2 directory:

```
cd /plonky2
Run docker:
docker build -t plonky2 .
docker run -it –rm plonky2
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run –bin cubic_expression
cargo run –bin range_proof
cargo run –bin sha256
```

Since plonky2 does not provide gadgets related to sha256 circuit construction, we seek an open source implementation of the forked plonky2 library and place it in our plonky2sha256 directory. If you want to run Sha256, navigate to the /plonky2sha256 directory:

```
cd /plonky2sha256
```

Navigate to the directory of the program you would like to run. Our examples are at /root/src. Run the following code to execute the examples.

```
cargo run
```

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.