USENIX Security '25 Artifact Appendix: Scalable Collaborative zk-SNARK and Its Application to Fully Distributed Proof Delegation

Xuanming Liu[1], Zhelei Zhou[1], Yinghao Wang[1], Yanxin Pang[3], Jinye He[4],
Bingsheng Zhang[1], Xiaohu Yang[1], Jiaheng Zhang[2]
[1]*Zhejiang University* [2]*National University of Singapore* [3]*Tsinghua University* [4]*University of Virginia*
{hinsliu, zl_zhou, asternight, bingsheng, yangxh}@zju.edu.cn,
jhzhang@nus.edu.sg, pangyx21@mails.tsinghua.edu.cn, qfn5bh@virginia.edu

A Artifact Appendix

A.1 Abstract

In a nutshell, this work enables many low-resource servers to jointly execute a *multiparty computation* (MPC) protocol for zk-SNARK proof generation. During the proof generation, the servers collaboratively compute a zk-SNARK proof for a large circuit while keeping their secret inputs—known as *the witness* in zk-SNARK terminology—hidden from each other. A key feature of our design is its *scalability*: each server incurs less computational and memory overhead compared to the original monolithic prover.

This artifact provides a proof-of-concept implementation. It includes a Rust-based prototype, supporting collaborative proof generation. The artifact includes implementations of collaborative primitives, the packed secret sharing scheme, and the complete collaborative zk-SNARK (HyperPlonk), which are introduced in the paper. Moreover, it provides two modes of execution: (i) Local mode and (ii) Distributed mode, allowing users to simulate the protocol either on a single machine or in a distributed network across multiple machines. Refer to the `README.md` file for detailed instructions on how to run the artifact. It also includes scripts to benchmark performance under various deployment configurations.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact is an academic prototype and is not suitable for production uses. It has not undergone formal reviews.

Although it involves cryptographic protocols and peer-to-peer communication, it does not perform any destructive or malicious operations on developers' machines. However, since the artifact requires network-based interactions between multiple parties, we recommend running it in a controlled environment, virtual machine or sandbox to mitigate potential risks related to system security or data privacy during testing.

A.2.2 How to access

The artifact is maintained at GitHub. A branch named `artifact-eval`, containing the code and docs specifically prepared for AE, is available at: https://github.com/LBruyne/Scalable-Collaborative-zkSNARK/tree/artifact-eval.

For artifact evaluation, we also provide a stable reference to the evaluated version. The direct link to this version is available on Zenodo: https://doi.org/10.5281/zenodo.16722573. We recommend reviewers download the artifact from the Zenodo link, as it includes additional documentation and detailed instructions for usage.

A.2.3 Hardware dependencies

This artifact does not require special hardware features.

As detailed in the `README.md` file, the artifact supports two modes of execution. Reviewer/User could choose the mode that best fits their testing environment and available resources.

- **Distributed execution mode (Benchmark):** This mode really runs a distributed network and supports large-scale deployment across 16-128 virtual or physical machines to run the collaborative proof generation. Each peer can be a low-resource instance with only 4GB of memory. All peers must be connected over a LAN or WAN network.
- **Local execution mode (Local/Leader):** (Recommended for its simplicity) Suitable for small-scale testing, this mode runs on a single machine. It does not require any special hardware beyond a standard system with sufficient computational resources. For reproducibility concerns, we recommend a machine with at least 1TB of RAM.

A.2.4 Software dependencies

This artifact is developed using the Rust nightly toolchain. The following specific versions were used for development and evaluation (but other versions may also work):

- `rustup` 1.27.1 (54dd3d00f 2024-04-24)

- `cargo` 1.80.0-nightly (05364cb2f 2024-05-03)

For convenience, we recommend installing the utility tool `just`, which simplifies execution and scripting. It can be installed from https://github.com/casey/just.

A Linux-based operating system is recommended. All development and testing were conducted on Ubuntu 22.04 LTS.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

First, download the artifact and install the required Rust nightly toolchain. We also recommend installing the `just` tool to simplify command execution.

Then, set up the environment with the following command:

```
rustup default nightly-2024-05-03 # You can use a
    different nightly version, but this is the one
    we used for development.
```

Next, build the artifact using:

```
cargo build --release
```

This command compiles the Rust code and prepares the executable files for use.

Finally, run a basic test to ensure the artifact is set up correctly:

```
just run --release --example sumcheck -F leader -- --
    l 4 --n 10
```

If you do not have `just` installed, you can run the following command instead:

```
RUSTFLAGS="-Ctarget-cpu=native -Awarnings" cargo +
    nightly run --release --example <example_name> <
    args>
```

where `<example_name>` is the name of the example (e.g., `sumcheck`), and `<args>` are the relevant command-line arguments (e.g., `-F leader - -l 4 -n 10`).

You may need to use the following commands to allow the user's shell to run the artifact properly:

```
ulimit -HSn 65536
```

This command runs a simple example of the collaborative sumcheck with input size 2^{10} in **leader** mode (refer to the `README.md` file), verifying that the main components of the artifact are functioning correctly. The output looks like this:

```
Start: Local Sumcheck (thread ThreadId(1))
End: Local Sumcheck (thread ThreadId(1)) ...s
Start: Collaborative Sumcheck Leader (thread
    ThreadId(1))
...
End: Collaborative SumcheckProduct Leader (
    thread ThreadId(1)) ...s
Comm: (...)
```

A.3.2 Basic Test

To test the artifact's functionality, run:

```
just run --release --example hyperplonk -F local --
    --l 8 --n 16
```

This command executes a moderate-size example of the collaborative HyperPlonk protocol in **local** mode (refer to the `README.md` file), verifying that the artifact is functioning correctly. This simulates the workload of $64 = 8 \times l$ parties on a $2^{16} = 2^n$-gate circuit *one-party-by-one-party* using a *single thread*. The full network is emulated locally.

The expected output should resemble:

```
Start: Local HyperPlonk (thread ThreadId(1))
...
End: Local HyperPlonk (thread ThreadId(1)) ...s
Start: Local HyperPlonk++ (thread ThreadId(1))
...
End: Local HyperPlonk++ (thread ThreadId(1)) ...
    s
Start: Simulate Collaborative Hyperplonk++ (
    thread ThreadId(1))
...
Comm: (...)
Comm: (...)
End: Simulate Collaborative Hyperplonk++ (thread
     ThreadId(1)) ...s
```

This command typically takes around 8 minutes to complete. You can notice that according to the log file, there are actually three instances being run: local HyperPlonk, local HyperPlonk++, and collaborative HyperPlonk++. The runtime duration is the runtime sum of these three instances. If it does not run successfully, please refer to the `README.md` file for troubleshooting. A common issue is insufficient memory allocation for virtual machines—try a smaller value of n, or use a machine with more memory.

A.4 Evaluation workflow

A.4.1 Major Claims

The protocol enables scalable collaborative zk-SNARK:

(C1): Each peer in the collaborative zk-SNARK (in this paper, HyperPlonk) incurs only a fraction ($O(\frac{1}{N})$, where N is the number of servers) of the time and memory overhead compared to the monolithic prover. The communication is small. This is demonstrated in experiment (E1), corresponding to Section 5.2, 6.2, Figure 3, and Table 2, 3 of the paper.

A.4.2 Experiments

First, we introduce (this can also be found in the `README.md` file) that the artifact provides two modes of execution:

- **Distributed execution mode:** The Rust feature **Benchmark**, enabled by `-F benchmark` in the command, specifies the protocol is run in a real distributed network. When benchmarking using this mode (this is what we do in the experiments), you need sufficient servers or machines connected over a LAN/WAN network. In the `README.md` file, we provide very detailed scripts to set up such a network and run the benchmarks. Especially note that there are many files and paths needed to by renamed, which are listed in the `README.md` file.
- **Local execution mode:** However, we understand reviewers may not have access to a distributed network or sufficient machines. Therefore, we also provide a local execution mode, which simulates the protocol in a single machine. The Rust feature **Local**, specifies the protocol is simulated locally. In this mode, A single thread will run each peer's workload one-by-one in circulation. This mode does not require a network connection.

Next, we describe the recommended experiments:

(E1): [Collaborative vs. Monolithic Prover] [30 human-minutes + 24 compute-hours]: This experiment compares the performance of collaborative HyperPlonk against the monolithic prover on general circuits.

If you are running the benchmark in **the distributed mode**:

Preparation: Provision 128 virtual machines, each with 2 vCPUs and 4GB RAM, connected over a LAN or WAN network. Additionally, set up a separate machine as a *jump server* that can access all 128 VMs via SSH. Ensure the network connectivity is functional.

Execution: Follow the instructions in the `benchmark` section of the `README.md` file. The script we prepared will automatically handle the setup and execution.

Note that to run this experiment, you only need the first three lines in the `run_all.sh` file.

Results: The results for each cases will be written to the `./output` directory. Use the provided Jupyter notebook (`./hack/read_data.ipynb`) to convert logs into CSV for performance comparisons. We expect to observe that the data reflect the Figure 3 and Table 2 in the paper. Note that in different runs, the computation time may vary slightly due to the non-deterministic nature of the network and system load. However, the overall linear trends should remain consistent.

If you are running the benchmark in **the local mode**:

Preparation: Prepare a single machine with at least 1TB of memory. This machine will run all the peers' workloads in a single thread, simulating the distributed network *locally*.

Execution: Run `bash ./hack/bench_hyperplonk.sh`. This script will automatically run the collaborative HyperPlonk and the monolithic prover in **local** mode, simulating the workload of $8l$ peers on a 2^n-gate circuit using a *single thread* locally.

Results: The output will be written to the `output` directory. The expected results are similar to the Figure 3 and Table 2 in the paper. Remember that in **local** mode, the actual computation time should be divided by $N = 8l$ where N is the number of peers, to obtain the average time per peer. After this division, the results should be similar to the distributed mode. For example, in the case of $l = 4$ and $n = 16$, the output should be similar to the following:

```
Start: Local HyperPlonk (thread
    ThreadId(1))
   ...
End: Local HyperPlonk (thread
    ThreadId(1)) 52.132s
Start: Local HyperPlonk++ (thread
    ThreadId(1))
   ...
End: Local HyperPlonk++ (thread
    ThreadId(1)) 65.256s
Start: Simulate Collaborative
    Hyperplonk++ (thread ThreadId(1))
   ...
Comm: (2298526, 3666742)
Comm: (2298526, 3666742)
End: Simulate Collaborative
    Hyperplonk++ (thread ThreadId(1))
     190.622s
```

Then each peer's average time is $\frac{190.6s}{4\times 8} = 5.9s$, while the monolithic prover's time is $65.3s$. Therefore, the simulated speedup is $\frac{65.3s}{5.9s} \approx 11$, which is consistent with the claims in the paper.

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/.