

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Distributed Synthesis of Differentially Private Tabular Datasets

Yucheng Fu, *University of Virginia;* Tianyao Gu and Elaine Shi, *Carnegie Mellon University;* Tianhao Wang, *University of Virginia*

https://www.usenix.org/conference/usenixsecurity26/presentation/fu

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology



USENIX Security '26 Artifact Appendix: Distributed Synthesis of Differentially Private Tabular Datasets

Yucheng Fu*, Tianyao Gu†, Elaine Shi†, Tianhao Wang*
*University of Virginia, †Carnegie Mellon University
{zdp8uu, tianhao}@virginia.edu, tianyaog@andrew.cmu.edu, elaineshi@cmu.edu

A Artifact Appendix

A.1 Abstract

This artifact is the implementation of the paper *Distributed Synthesis of Differentially Private Tabular Datasets*. Our goal with the artifacts is to provide the source code and scripts to realize the claimed functionality described in the paper, and reproduce the experimental results.

In our paper, we proposed two primitives to facilitate the differentially private tabular synthesis in the distributed setup: a two-way marginal computation protocol and a noise generation protocol. Correspondingly, our artifact contains a separate implementation of these two primitives that might be of independent interest in other applications. We also provide the implementation of our end-to-end synthesis protocol that utilizes these primitives.

We provide all our experiment code either as one-line commands or as a simple script file to make the results simpler to reproduce. Our main goal with the experiments is not to reproduce the exact same numbers as we have in our figures, as it will vary greatly with the hardware used, but to show that the speedup we obtain, as well as the asymptotic behavior, is similar to the shown in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. Our code does not have any destructive steps, nor does it disable any security mechanisms.

A.2.2 How to access

Our artifact is available in Zenodo: https://doi.org/10.5281/zenodo.18218427

A.2.3 Hardware dependencies

In terms of infrastructure, all of our experiments were run on the same AWS instance type c5a.8xlarge, with 32 CPU of x86_64 architecture, 64 GiB RAM, and 64 GiB storage. The OS is Ubuntu Server 22.04 LTS (HVM), SSD Volume Type. We suggest using a similar OS and at least 32 GB free disk space. Otherwise, the automatic environment setup script might fail.

A.2.4 Software dependencies

Our artifact does not depend on any pre-installed software. All dependencies (Conda, Docker, Python environments, etc.) are automatically installed by the setup script, and most of the experiments are meant to be run under Docker.

A.2.5 Benchmarks

We use the UCI Adult dataset https://archive.ics.uci.edu/dataset/2/adult, which is licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) license: https://creativecommons.org/licenses/by/4.0/legalcode.

A.3 Set-up

A.3.1 Installation

Use terminal to navigate into the root path of the artifact. We provide a single-line command to set up the environment.

```
./run_exp.sh setup
```

Running the script "./run_exp.sh" given the "setup" parameter will install all the dependencies of our experiment, including Conda, Docker, and the necessary Python environment, etc. This script does not assume any pre-installed package and can make sure the later experiments can be run correctly even in a newly initialized instance.

A.3.2 Basic Test

After setup, start the Docker containers and navigate to the artifact root directory:

```
cd docker && sudo docker-compose up -d && cd ..
```

We provide two lightweight test scripts to verify the installation without running the full experiments.

Test 1: Marginal computation. (test_marginal.sh) This script runs the two-way marginal computation on the default demo dataset (`adult_small.csv`) through the full DPF protocol: key generation on the client, key evaluation on 3 parties, share collection, and reconstruction:

```
./test_marginal.sh
```

Expected output upon success:

```
==========================================
TEST PASSED
  - Marginal shares collected from 3 parties
  - Reconstruction successful
  - Total sum = 100 (matches 100 records)
==========================================
```

Test 2: Secure noise generation (test_noise.sh). This script runs a single secure noise generation protocol across 3 parties using MP-SPDZ. It accepts three optional arguments: noise type, number of samples, and scale parameter.

```
./test_noise.sh [dgauss|dlap] [n_samples]
                [scale]
```

The defaults are `dgauss`, `4096`, and `10`, respectively. To quickly test with the default parameters:

```
./test_noise.sh
```

Expected output upon success:

```
==========================================
TEST PASSED
  - Noise generation completed successfully
  - Time: <time>s
  - Generated <n> noise samples
==========================================
```

Note: Both tests require Docker containers to be running (set up using `./run_exp.sh setup`). Results and logs are saved to `test_result/`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our two-way marginal computation (joint histogram) protocol achieves at least a four-order-of-magnitude improvement in communication and a five-order-of-magnitude improvement in running time compared with the existing protocol, CaPS. Compared with the alternative method Sort-count, we also make $122\times \sim 1218\times$ improvement in communication and $30\times \sim 290\times$ improvement in running time. This is demonstrated by experiment (E1), with results shown in Figure 8.

(C2): Our secure noise generation protocol (discrete Gaussian and discrete Laplace) achieves significantly smaller running time compared to the previous protocols. Our improvement primarily lies in the communication round, which is suitable for a high-latency network environment. This is demonstrated by experiment (E2), with results shown in Figure 9 and Table 4.

(C3): Our end-to-end distributed synthesis protocol can generate differentially private synthetic datasets with practical efficiency. For example, we can synthesize the "Adult" dataset in 24 minutes in a real-world WAN setting. This is demonstrated by experiment (E3), with results shown in Table 5. Using our discrete noise implementation, we can still achieve a comparable utility (ML accuracy and query error) as the centralized DP data synthesis algorithm. This is demonstrated by experiment (E4), with results shown in Table 6.

A.4.2 Experiments

Most of our experiments are executed on four Docker containers. One acts as the client to generate all the DPF keys and sends these keys to three other containers acting as the three servers. We suggest using the following command to run all the experiments:

```
nohup ./run_exp.sh all > exp.log 2>&1 &
```

It will take about 7 hours to finish all the experiments in our paper. After finishing, all the numerical results will be saved to the path `result/` and all the graphs in our paper will be saved to the path `result_figures/`, which should contain the following files:

```
result_figures/
  Figure8_marginal_cost.pdf
  Figure8_marginal_cost.png
  Figure9_noise_time.pdf
  Figure9_noise_time.png
  Table4_detailed_noise.csv
  Table5_end2end_generation.csv
  Table6_utility_comparison.csv
  Table6_utility_comparison.tex
```

The names of these files correspond to the figure and table numbers in our paper. Next, we describe each experiment in detail and provide an individual command for each experiment.

(E1): Two-Way Marginal Computation [1 human-minute + 2 compute-hours + 32GB disk]: Compare our DPF-based two-way marginal computation protocol against two baselines (Sort-count and CaPs) across varying numbers of records n and domain sizes u. This supports claim (C1), with results shown in Figure 8 in our paper.

How to: Run the marginal comparison script. The script benchmarks three methods (Our Method, Sort-count,

CaPs) by varying n from 1,000 to 10,000 (with fixed $u = 100$) and varying u from 20 to 200 (with fixed $n = 10{,}000$). For each configuration, it records wall-clock time (seconds) and communication cost (MB). After the benchmarks complete, it automatically generates Figure 8 using the plotting environment.

Preparation: Ensure the Docker containers (`client`, `party0`, `party1`, `party2`) are running. The environments for DPF (via Bazel) and the MP-SPDZ environment have already been configured during installation.

Execution: If you have run "`nohup ./run_exp.sh all`", then no further action needs to be taken. To run this experiment individually, use command:

```
./run_exp.sh marginal
```

Results: All the raw CSV data will be saved as:
`result/vary_n_time.csv`,
`result/vary_n_comm.csv`,
`result/vary_u_time.csv`,
`result/vary_u_comm.csv`.
Each CSV contains 3 rows corresponding to our method, `sort_count`, and `caps`. The Figure 8 in our paper is:
`result_figures/Figure8_marginal_cost.pdf`

(E2): Noise Generation Comparison [1 human-minute + 2 compute-hours + 32GB disk]: This experiment evaluates our CDF-based distributed noise generation protocols (CDF-dGauss, CDF-dLap) against the ODO baselines (ODO-dGauss, ODO-dLap) for discrete Gaussian and discrete Laplace distributions. It supports claim (C2), with results shown in Figure 9 and Table 4 in our paper.

How to: The script has two parts: (1) varying the number of samples $n \in \{4, 16, 64, 256, 1024, 4096, 16384\}$ with fixed scale $= 10$, and (2) varying scale $\in \{1, 5, 10, 15, 20, 25, 30\}$ with fixed $n = 1024$. For each configuration and method, it records wall-clock time (seconds), communication cost (MB), and number of rounds. It also runs a detailed comparison with $n = 16384$ and scale $= 30$. After all benchmarks are complete, it generates Figure 9 and Table 4.

Preparation: Ensure the Docker containers (`party0`, `party1`, `party2`) are running. The MP-SPDZ environment (libraries, SSL certificates, IP file) has already been configured during installation.

Execution: If you have run "`nohup ./run_exp.sh all`", then no further action needs to be taken. To run this experiment individually:

```
./run_exp.sh noise
```

Results: All the raw CSV data will be saved as
`result/noise_vary_n_time.csv`,
`result/noise_vary_n_comm.csv`,
`result/noise_vary_n_rounds.csv`,
`result/noise_vary_scale_time.csv`,
`result/noise_vary_scale_comm.csv`,
`result/noise_vary_scale_rounds.csv`.
Each CSV contains 4 rows corresponding to CDF-dGauss, CDF-dLap, ODO-dGauss, and ODO-dLap.
The Table 4 data in our paper is saved as:
`result_figures/Table4_detailed_noise.csv`
The Figure 9 in our paper is saved as:
`result_figures/Figure9_noise_time.pdf`

(E3): End-to-End TDS Generation [1 human-minute + 30 compute-minutes + 32GB disk]: Run the distributed AIM synthesis pipeline on the UCI Adult dataset, combining DPF-based marginal computation with general MPC-based selection and measurement. This supports claim (C3), as shown in Table 5 of our paper.

How to: Run the end-to-end generation script. The script first performs DPF marginal computation (keyGen on the client, keyEval on 3 parties), then runs the distributed AIM synthesis (`dist_aim_mpc.py`) with 35 AIM rounds, $\varepsilon = 1.0$, and $\delta = 10^{-9}$. The DPF phase runs first, and its timing is passed to the synthesis phase for combined reporting of per-phase breakdowns (marginal computation, selection, measurement) and total time/communication.

Preparation: Ensure four Docker containers (`client`, `party0`, `party1`, `party2`) are running and the Conda environment `dp_tds_plot` is available. Both have already been configured during installation.

Execution: If you have run "`nohup ./run_exp.sh all`", then no further action needs to be taken. To run this experiment individually:

```
./run_exp.sh tds
```

Results: The Table 5 data in our paper is saved as:
`result_figures/Table5_end2end_generation.csv`
This file contains per-phase breakdowns (marginal computation, selection, measurement, generation) and total time/communication.

(E4): Utility Comparison [5 human-minutes + 30 compute-minutes + 32GB disk]: Compare the downstream ML utility and query accuracy of synthetic data generated by standard (centralized) AIM versus our distributed AIM implementation. This supports claim (C3), with results shown in Table 6 in our paper.

How to: Run the utility comparison script. The script runs both standard AIM (continuous Gaussian noise + exponential mechanism) and simulated distributed AIM (discrete Gaussian noise + report-noisy-max) on the Adult dataset for each privacy budget $\varepsilon \in \{0.1, 0.5, 1.0, 5.0, 10.0\}$. For each run, it trains downstream ML classifiers (logistic regression, decision tree, random forest, MLP, XGBoost, CatBoost) on the generated synthetic data and evaluates on real test data, recording accuracy, F1 score, and AUC. It also computes the

conditional query error between the synthetic and real data distributions.

Preparation: Ensure the `tab_bench` conda environment is available. It has already been configured during installation. This experiment does not require Docker containers.

Execution: If you have run "`nohup ./run_exp.sh all`", then no further action needs to be taken. To run this experiment individually:

./run_exp.sh utility

Results: The Table 6 data in our paper is saved as: `result_figures/Table6_utility_comparison.csv` This file contains per-ε accuracy, F1 score, AUC, and query error for both standard and distributed AIM. A LaTeX formatted table is also generated at: `result_figures/Table6_utility_comparison.tex` Distributed AIM should achieve similar accuracy, F1 score, and query error as standard AIM across all tested privacy budgets.

A.5 Notes on Reusability

Our artifact is designed to be modular and reusable:

Using individual primitives: The two-way marginal computation based on three-party DPF and noise generation protocols can be used independently in other MPC applications. Their source codes are located in `src/3party-dpf/` and `src/libs/disGauss.py` with clear interfaces.

In particular, we present the first implementation of a three-party DPF that supports local multiplication. This is especially helpful to the data analysis in a vertically distributed setup.

Adjusting parameters:

- Network conditions: Use `./docker/setup_env.sh wan [latency_ms] [bandwidth_mbit]` to simulate different network settings
- Other datasets: By modifying the marginal computation protocol in `src/marginal/`, and changing the parameters in `src/marginal/libs/e2e/dist_aim_mpc.py`, the generation protocol can support other datasets.

Extending the protocols: The MPC protocols are implemented using a combination of secret sharing and function secret sharing. Researchers can extend the protocols by:

- Adding new noise generation protocols in `src/noise/`.
- Improving the marginal computation protocol (three-party DPF) in `src/3party-dpf/`, including extending our security model to malicious security or supporting a larger number of parties.
- Integrating with other synthesis algorithms beyond AIM.

A.6 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/usenixsec2026/.