

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

SING: Improving the Efficiency of MPC Protocol Assignment using Graph Neural Networks

Jannis Blüml, Moritz Huppert, Nora Khayata, Joachim Schmidt, and Thomas Schneider, *Technical University of Darmstadt*

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

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: SING: Improving the Efficiency of MPC Protocol Assignment using Graph Neural Networks

Jannis Blüml[1] Moritz Huppert[1] Nora Khayata[2] Joachim Schmidt[2] Thomas Schneider[2]
Technical University of Darmstadt, Darmstadt, Germany
[1]{firstname}.{lastname}@tu-darmstadt.de
[2]{khayata,joachim.schmidt,schneider}@encrypto.cs.tu-darmstadt.de

A Artifact Appendix

A.1 Abstract

The artifact consists of the codebase and datasets accompanying SING, a machine-learning-based framework for hybrid MPC share assignment. Specifically, we provide three archives: (i) the complete source directory tree of our codebase, including the pretrained models, (ii) the full unprocessed dataset of MPC circuits in C source code format, and (iii) the fully processed dataset of MPC circuits. The codebase includes a detailed `README.md` with installation and reproduction instructions.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

We release a permanently accessible version of our artifact on Zenodo[1]. This release comprises all code used as well as the complete dataset used to train our models. Additionally, we release our code on GitHub[2]. During the evaluation period, we will continue to update the GitHub repository as needed. Once the evaluation phase has concluded, we will (permanently) incorporate all changes into the Zenodo repository.

A.2.3 Hardware dependencies

Evaluating our pretrained share-assignment and cost-prediction models on reasonably small circuits does not require specialized hardware. Model inference runs on the CPU by default, although larger circuits require more RAM. We recommend at least 4 GB of free RAM for evaluating our benchmark circuits. Training the models from scratch, rather than using the provided pretrained models for inference, requires hardware acceleration. For training on the full dataset, we recommend a GPU with at least 48 GB of VRAM.

A.2.4 Software dependencies

We recommend running all experiments on Debian 13 or Ubuntu 26.04. While the model training and inference pipelines of SING are OS-agnostic, the external dependencies ABY [4] and Silph [3] are supported only on Linux. We note that for training and evaluating our share assignment and cost prediction models based on the existing (full) dataset, these third-party dependencies are not required. However, generating new data and benchmarking requires ABY to benchmark circuit runtimes and Silph to generate optimal assignments. Furthermore, benchmarking MPC runtime and communication using our benchmark harness requires root privileges in order to simulate different network environments.

SING Training and evaluation of our share assignment and cost prediction models requires a Python $\geq$ 3.13 installation. These packages are expected to be installed:

```
torch          torchvision
torchaudio     torcheval
torchmetrics   torch_geometric
tqdm           line_profiler
black          tabulate
wandb          pandas
plotly         kaleido
dash           scipy
```

Silph The following Debian packages are required:

```
curl           cmake
coinor-cbc     coinor-libcbc-dev
cvc4           git
libgmp-dev     libboost-all-dev
libssl-dev     time
zsh            build-essential
python3        python3-pip
```

[1] https://doi.org/10.5281/zenodo.20347718
[2] https://github.com/encryptogroup/SING

ABY The following Debian packages are required:

```
build-essential   cmake
libgmp-dev        libssl-dev
doxygen           libboost-all-dev
```

A.2.5 Benchmarks

All 9 benchmark circuits used in the paper are included in the full dataset available in our Zenodo repository (`dataset-full.tar.gz`). We list the circuit files and hashes used in the `benchmark-results/circuits.csv` file.

A.3 Set-up

First, download the (up to date) code archive from GitHub (`sing-artifact.tar.gz`) and the processed full dataset from Zenodo (`dataset-full.tar.gz`). As mentioned in Section A.2.2, we will update the GitHub repository with patches and fixes throughout the evaluation process. While the code repository (GitHub) includes a small excerpt of the dataset (`dataset-excerpt.tar.gz`), reproducing model training from scratch and our benchmarks requires the full dataset, which is hosted only on Zenodo due to its size.

A.3.1 Installation

SING We recommend setting up an isolated virtual environment and installing the packages listed in section A.2.4 in this environment. Run the following commands in the root of the code repository:

```
python -m venv venv
source venv/bin/activate

pip install torch torchvision torchaudio

pip install -r requirements.txt
```

The installation of PyTorch [1] may differ depending on your hardware. We refer to the official documentation[3] for further instructions.

ABY We have prepared a `Dockerfile` in the `ABY-vendor` subdirectory to provide a reproducible build process in an isolated environment.

```
docker build -t aby .
```

Alternatively, the build steps specified in the `Dockerfile` can be run on the host instead. Run the following commands in the `ABY-vendor` subdirectory of the code repository:

```
apt-get install \
build-essential \
cmake \
libgmp-dev \
libssl-dev \
doxygen \
libboost-all-dev

mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release \
-DABY_BUILD_EXE=On && make
```

Depending on the CMake version, it may be necessary to add the following flag to the `cmake` invocation: `-DCMAKE_POLICY_VERSION_MINIMUM=3.5`.

Silph We have prepared a `Dockerfile` in the `silph` subdirectory to provide a reproducible build process in an isolated environment.

```
docker build -t silph .
```

Alternatively, the build steps specified in the `Dockerfile` can be run on the host instead. Afterward, run the following commands in the `silph` subdirectory of the code repository:

```
apt-get install \
curl \
cmake \
coinor-cbc \
coinor-libcbc-dev \
cvc4 \
git \
libgmp-dev \
libboost-all-dev \
libssl-dev \
time \
zsh \
build-essential \
python3 \
python3-pip

curl --proto '=https' --tlsv1.2 \
-sSf https://sh.rustup.rs | sh -s -- -y

source "$HOME/.cargo/env"

python3 driver.py --all_features
mkdir -p scripts/aby_tests/tests
python3 driver.py -c
python3 driver.py -i
python3 driver.py -b
```

A.3.2 Basic Test

To test the basic functionality of SING, we perform share assignment on an MPC circuit and subsequently execute this circuit with ABY [4].

Performing share assignment First, we use the S^{SING} model with pretrained weights to generate a share assign-

[3] https://pytorch.org/get-started/locally/

ment for the Gauss benchmark. Run the following command in the root of the code repository:

```
python inference.py --checkpoint \
pretrained/share_assignment_supervised.pt \
--output share_assignment.txt \
example-gauss
```

The resulting share assignment computed by the model will be written to `share_assignment.txt`.

Executing a mixed-protocol MPC circuit Second, we execute the circuit with the generated share assignment in a mixed-protocol MPC setting with ABY [4]. We run both parties locally on the same host for simplicity.

Using ABY built with a Docker container (cf. section A.3.1): Run the following command in the repository root:

```
docker run --rm -it \
-v ./example-gauss:/circuit \
-v ./share_assignment.txt:/share_assignment.txt \
aby
```

Using ABY built on the host: Run the following command in the repository root:

```
BINARY=ABY-vendor/build/bin/aby_interpreter
CIRCUIT_DIR=example-gauss
SHARE_ASSIGNMENT_FILE=share_assignment.txt

# server
"$BINARY" -m mpc -f "$CIRCUIT_DIR" \
-s "$SHARE_ASSIGNMENT_FILE" -r 0 &

# client
"$BINARY" -m mpc -f "$CIRCUIT_DIR" \
-s "$SHARE_ASSIGNMENT_FILE" -r 1 &
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The share assignment runtime of SING outperforms the current state-of-the art Silph [3]. This is proven by experiment (E1) whose results are reported in Table 2 of our paper.

(C2): The MPC performance of share assignments generated by SING is similar to the current state-of-the-art Silph [3]. This is proven by experiment (E2) whose results are reported in Tables 6, 7, and 8 of our paper.

(C3): Training our models from scratch using ground-truth labels from Silph [3], computed Silph costs, or collected MPC performance measurements is efficient. This is proven by experiment (E3) whose results are reported in Table 3 of our paper.

(C4): Our models consider both the circuit structure and the operation types of individual nodes when approximating costs or assigning shares. This is proven by experiment (E4) whose results are discussed in section 8.7 of our paper.

A.4.2 Experiments

(E1): Share assignment benchmarks [5 human-minutes + 24 compute-hours + 5GB disk]: We compare the share assignment runtimes of Silph [3] and SING for the 9 benchmark circuits named in our paper.
Preparation: Compile the Silph framework without Docker and set up the Python virtual environment with all dependencies (cf. section A.3.1). Download the full processed dataset (`dataset-full.tar.gz`) from Zenodo and extract it in a suitable location. We refer to the extracted location by `$DATASET`.
Execution: Run the following command in the repository root:

```
python generate_dataset_split.py \
--dataset-dir $DATASET

mkdir dataset || rm -rf dataset \
&& mkdir dataset

ln -s $DATASET dataset/raw

# Silph without CSS
python benchmark_share_assignment.py \
--circuits-file paper_benchmark_c.txt \
--timeout 40000

# Silph with CSS
python benchmark_share_assignment.py \
--silph-selection-scheme css \
--circuits-file paper_benchmark_c.txt \
--timeout 40000 \
--output-csv share_assignment_css.csv
```

Results: The collected share assignment runtimes will be written to the files `share_assignment.csv` and `share_assignment_css.csv` for comparison with Silph and Silph+CSS respectively. These results can be compared with our results in the `benchmark-results/share-assignment` subdirectory. Specific result numbers depend on hardware and software configuration.

(E2): MPC benchmarks [5 human-minutes + 1.5 compute-hours + 5GB disk]: We run the 9 benchmark circuits named in our paper in a mixed-protocol MPC setting with share assignments either generated by Silph [3] or by SING and compare the results.
Preparation: Build the ABY framework without Docker and set up the Python virtual environment with all dependencies (cf. section A.3.1). Download the full processed dataset (`dataset-full.tar.gz`) from Zenodo and extract it in a suitable location. We refer to the extracted location by `$DATASET`.
Execution: Run the following commands in the repository root:

```
python generate_dataset_split.py \
--dataset-dir $DATASET

mkdir dataset || rm -rf dataset \
&& mkdir dataset

ln -s $DATASET dataset/raw

scripts/benchmark_combinations.sh
```

Results: The benchmark results will be written to different CSV files of the format `<MODEL_UUID>_<NETWORK_SETTING>.csv`. These CSV files can be compared with our results in the `benchmark-results/mpc` subdirectory. The mapping of model UUID to model variant (i.e., SING 1, 2, 3) is specified in `benchmark-results/plot.py`. Specific result numbers depend on hardware and software configuration.

We note that this experiment uses models trained with the clean training pipeline, which explicitly excludes the Silph benchmark circuits from the training dataset. We briefly describe this clean training pipeline below. The pipeline follows the same steps as E3 but uses slightly altered hyperparameters and specifically excludes the Silph benchmark circuits using the `-exclude-file` option with `exclude-silph.txt`.

```
wandb disabled
python generate_dataset_split.py \
--exclude-file exclude-silph.txt \
--alpha 0.5 \
--dataset-dir $DATASET

mkdir dataset || rm -rf dataset \
&& mkdir dataset

ln -s $DATASET dataset/raw

# Share assignment: SING 1
time python train.py --lr 0.01 --alpha 0.9

# Share assignment: SING 2
time python train.py --lr 0.001 --alpha 0.1 \
--predicted-cost --checkpoint \
pretrained/clean/cost_prediction_silph.pt

# Share assignment: SING 3
time python train.py --lr 0.001 --alpha 0.99 \
--predicted-cost --checkpoint \
pretrained/clean/cost_prediction_runtime.pt

mkdir dataset-cost-prediction \
|| rm -rf dataset-cost-prediction \
&& mkdir dataset-cost-prediction

ln -s $DATASET dataset-cost-prediction/raw

# Cost prediction: Train with Silph cost
time python train_cost_prediction.py \
--lr 0.001 --cost-name silph \
--assignment-filter no_all

# Cost prediction: Train with MPC runtime
time python train_cost_prediction.py \
--lr 0.001 --cost-name runtime-neon-wanfast \
--assignment-filter spn
```

(E3): Training our models [5 human-minutes + 1 compute-hour + 5GB disk]: We train our S^{SING} models *from scratch* in a supervised manner (SING 1) and in a semi-supervised manner (SING 2, SING 3). This training uses the pre-trained cost models C^{SING}. In addition, we train C^{SING} *from scratch* to approximate the Silph cost model and real runtime costs.

Preparation: Set up the Python virtual environment with all dependencies (cf. section A.3.1). Download the full processed dataset (`dataset-full.tar.gz`) from zenodo and extract it in a suitable location. We refer to the extracted location by `$DATASET`.

Execution: Run the following commands in the repository root:

```
wandb disabled
python generate_dataset_split.py \
--dataset-dir $DATASET

mkdir dataset || rm -rf dataset \
&& mkdir dataset

ln -s $DATASET dataset/raw

# Share assignment: SING 1
time python train.py --lr 0.01 --alpha 0.9

# Share assignment: SING 2
time python train.py --lr 0.001 --alpha 0.1 \
--predicted-cost --checkpoint \
pretrained/cost_prediction_silph.pt

# Share assignment: SING 3
time python train.py --lr 0.01 --alpha 0.9 \
--predicted-cost --checkpoint \
pretrained/cost_prediction_runtime.pt

mkdir dataset-cost-prediction \
|| rm -rf dataset-cost-prediction \
&& mkdir dataset-cost-prediction

ln -s $DATASET dataset-cost-prediction/raw

# Cost prediction: Train with Silph cost
time python train_cost_prediction.py \
--lr 0.001 --cost-name silph \
--assignment-filter silph_and_random

# Cost prediction: Train with MPC runtime
time python train_cost_prediction.py \
--lr 0.001 --cost-name runtime-neon-wanfast \
--assignment-filter silph_and_perturb
```

Results: The last five commands will print out the execution time of the respective training run. These can be compared with our results reported in Table 3 of our paper. Specific result numbers depend on hardware and software configuration.

(E4): Ablation [5 human-minutes + 1 compute-hour + 5GB disk]: We train our S^{SING} and C^{SING} models identically to (E3), with the exception that we apply two types of ablation: removing all edges and zeroing out all node features.
Preparation: Identical to preparation of (E3).
Execution: Run the following commands in the repository root:

```
wandb disabled
python generate_dataset_split.py \
--dataset-dir $DATASET

mkdir dataset || rm -rf dataset \
&& mkdir dataset

ln -s $DATASET dataset/raw

# Share assignment: normal
python train.py --lr 0.01 --alpha 0.9

python eval.py
python benchmark_mpc.py --mode benchmark \
--hashes-file ablation_hashes.txt \
--network-setting LAN \
--output-csv normal.csv \
--timeout 300

# Share assignment: remove all edges
python train.py --lr 0.01 --alpha 0.9 \
--delete-edges

python eval.py
python benchmark_mpc.py --mode benchmark \
--hashes-file ablation_hashes.txt \
--network-setting LAN \
--output-csv delete-edges.csv \
--timeout 300

# Share assignment: zero out node features
python train.py --lr 0.01 --alpha 0.9 \
--zero-features

python eval.py
python benchmark_mpc.py --mode benchmark \
--hashes-file ablation_hashes.txt \
--network-setting LAN \
--output-csv zero-features.csv \
--timeout 300

mkdir dataset-cost-prediction \
|| rm -rf dataset-cost-prediction \
&& mkdir dataset-cost-prediction

ln -s $DATASET dataset-cost-prediction/raw



# Cost prediction: remove all edges
python train_cost_prediction.py \
--lr 0.001 --cost-name silph \
--assignment-filter silph_and_random \
--delete-edges

python eval_cost_prediction.py

# Cost prediction: zero out node features
python train_cost_prediction.py \
--lr 0.001 --cost-name silph \
--assignment-filter silph_and_random \
--zero-features

python eval_cost_prediction.py
```

Results: The invocations of `eval.py` and `eval_cost_prediction.py` will print out accuracy (for S^{SING}) and mean-squared error metrics (for C^{SING}). The invocations of `benchmark_mpc.py` will output MPC runtimes for the ILP benchmark circuit mentioned in our paper in `normal.csv`, `delete-edges.csv`, and `zero-features.csv`. These results can be compared with the results we provide in `benchmark-results/ablation`. Specific result numbers depend on hardware and software configuration.

A.5 Notes on Reusability

Our framework can immediately be reused for fast hybrid share assignment using the provided pretrained models. Moreover, models can be retrained for new deployment settings or state-of-the-art protocol suites (e.g., [5]) by generating fresh training data from MPC runtime benchmarks and the associated optimal Silph assignments.

In the fully bootstrapped setting, SING 3 trains directly on real-world runtimes. We found this regime to be brittle: it generalized poorly, produced a negative R^2, and required substantial hyperparameter tuning and dataset refinement. Since SING 2 performed well with a simpler cost model, these results suggest that the runtime-based cost surrogate is the main bottleneck. Future work could improve this regime by constructing larger and more diverse datasets, thereby yielding a more reliable runtime surrogate. Additional directions include tighter integration with Silph [3], for example, by using S^{SING} to provide warm starts for ILP solving; evaluating stronger graph architectures, such as graph transformers [7]; or exploring alternative training paradigms, such as reinforcement learning or generative graph models [2, 6].

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/.

References

[1] Jason Ansel et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *ASPLOS*, 2024.

[2] Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. Netgan: Generating graphs via random walks. In *ICML*, 2018.

[3] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Wenting Zheng. Silph: A framework for scalable and accurate generation of hybrid MPC protocols. In *S&P*, 2023.

[4] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[5] Radhika Garg, Kang Yang, Jonathan Katz, and Xiao Wang. Scalable mixed-mode MPC. In *S&P*, 2024.

[6] Ian J. Goodfellow et al. Generative adversarial nets. In *NeurIPS*, 2014.

[7] Ladislav Rampásek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. In *NeurIPS*, 2022.