USENIX Security '26 Artifact Appendix: FLOSS: Fast Linear Online Secret-Shared Shuffling

Ian Chang
University of Washington

Sela Navot
University of Washington

Alex Ozdemir
Max Planck Institute for Security and Privacy, Georgia Tech

Nirvan Tyagi
University of Washington

A Artifact Appendix

A.1 Abstract

We present FLOSS, a protocol for fast online linear secret-shared shuffling and a framework to build arithmetic permutation circuits. Our artifact includes:

1. the computational and communication efficiency comparison of FLOSS against various shuffling baselines;
2. the computational and communication efficiency evaluation of FLOSS used in radix sort (Radix[FLOSS]) compared with other state-of-the-art sorting alternatives;

This artifact reproduces the tables and figures in our paper.

A.2 Description & Requirements

FLOSS enables efficient online shuffling in the two-party computation (2PC setting). We provide shuffling benchmarks for FLOSS, the constant round malicious secure permutation network by Mohassel et al. (PermNet), Oblivious Punctured Matrix from Song et al. (OPM), and the canonical $\log(n)$-round malicious-secure permutation network circuit from Waksman (SimplePermNet). We also provide a case-study applying our shuffling protocol to several sorting benchmarks. These sorting benchmarks include MP-SPDZ's sorting network (SortNet), MP-SPDZ quicksort combined with shuffling protocols (StS[FLOSS], StS[OPM], StS[PermNet], StS[SimplePermNet]), and radix-sort combined with shuffling protocols (Radix[FLOSS], Radix[PermNet], Radix[SimplePermNet]). We evaluate both the preprocessing time (MP-SPDZ preprocessing generation, any input-independent circuit computation) and the online time (any input-dependent circuit computation).

We highly recommend using AWS EC2 instances for the artifact evaluation, which requires approximately $150 cost in AWS credits. The minimal hardware requirements to run our full set of our benchmarks is two AWS c5.18xlarge instances (72 vCPUs and 144GB memory each), each running Ubuntu 22.04 LTS. We provide one-run scripts that automatically instantiates the AWS instances with required dependencies and benchmarks.

For reviewers without AWS access, we also support running the *full* benchmark suite, including the pareto plots, on local hardware comparable to the AWS setup: either two local Ubuntu servers (each with ≈72 hardware threads and 144GB memory, connected over a high-bandwidth network), or a single large local Ubuntu server (with ≈144 hardware threads and 288GB memory) partitioned into two Docker containers that emulate the two parties. We provide a Dockerfile and container scripts that automate installation and benchmarking for this setup.

Ubuntu 22.04 LTS (recommended) or Mac OS X Sequoia is required in order to run benchmarks on a single smaller local computer, which only runs a *subsample* of our benchmarks due to decreased compute power. We require that the user has at least the compute power of at least 16 GB of memory and an Apple M4 chip for CPU (or equivalent).

A.2.1 Security, privacy, and ethical concerns

FLOSS is a protocol for privacy-preserving computation. It can be used to compute any function over user data in the malicious-secure two-party setting.

Note that there are always challenges when migrating from an existing system to a privacy-compliant one. MPC-based systems are still very new, and they are not easy to implement correctly. As a result, companies and organizations should think carefully about the privacy implications of the statistics they are computing over user data and be transparent about what aggregated data is being computed.

A.2.2 How to access

FLOSS is available here. Reviewers should download and extract the Zenodo archive, and then run all commands be-

low unless otherwise stated. The full artifact documentation, including `ARTIFACTS.md`, is included in the Zenodo archive.

We have also prepared scripts to automatically launch clusters and run the experiments, given that the reviewer has access to AWS EC2 instances, as well as Docker-based scripts for reviewers who instead have access to comparable local server hardware. The essential setup and reproduction steps for all options are included directly below.

A.2.3 Hardware dependencies

FLOSS does not require any specialized hardware. Our experiments were performed on AWS c5.18xlarge instances, each with 72 vCPUs and 144 GB memory. Reviewers without AWS access can reproduce the full benchmark suite on comparable local hardware instead: either two local Ubuntu servers, each with approximately 72 vCPUs and 144GB memory, or a single large local Ubuntu server with at least 144 vCPUs and 288GB memory, which we partition into two Docker containers of 72 vCPUs and 144GB memory each to emulate the two parties. Different hardware configurations will affect the concrete performance of FLOSS, but will result in a similar performance gain over the baselines.

A.2.4 Software dependencies

Option 1 (AWS EC2, recommended, full results). The one-run scripts in `artifacts/` automatically set up the AMI cluster, install prerequisites on the remote machines, execute the full benchmark suite, and collect the results. The local machine used to control the AWS run needs Python 3, the Python plotting libraries, TeX Live/MacTeX with `pgf` and `xcolor`, AWS CLI version 2, and AWS credentials configured for region `us-west-2` with JSON output.

Option 2 (large local server, no AWS, full results). This option reproduces the full benchmark suite, including the pareto plots, without AWS access, using two local Ubuntu servers or a single large local Ubuntu server running Docker. We provide `artifacts/local-large.Dockerfile`, which clones FLOSS, installs the OS, Python, Rust, and MP-SPDZ dependencies (including the `-DINSECURE` MP-SPDZ build described below), installs OPM's dependencies, and builds FLOSS inside the image. The local server needs Docker installed with enough free vCPUs and memory to allocate 72 vCPUS and 144GB, respectively, to each of two containers (288GB total) if using the single-server Docker setup; the two-server setup instead needs FLOSS and its dependencies installed natively on each server, as in Option 3 below.

Option 3 (single local computer, subsampled results). To install FLOSS only on a smaller local machine, the OS required is either Ubuntu 22.04 LTS or Mac OS X (Sequoia 15.7.4 works to our knowledge). This workflow only runs a subsample of the experiment plots, so we require that the user has at least 16GB RAM and an Apple M4 chip or equivalent. The user must set up the following dependencies:

1. Install necessary libraries (Ubuntu): `sudo apt-get install automake build-essential clang cmake git libboost-dev libboost-filesystem-dev libboost-iostreams-dev libboost-thread-dev libgmp-dev libntl-dev libsodium-dev libssl-dev libtool python3 unzip`
2. Install necessary libraries (Mac OS X): `brew install automake cmake boost gmp ntl libsodium openssl libtool python unzip`
3. Install Python libraries: `python3 -m pip install numpy matplotlib pandas scipy latex`
4. Install TexLive/MacTeX: https://tug.org/texlive/
5. Install TexLive/MacTeX libraries: `sudo tlmgr update -self; sudo tlmgr install pgf; sudo tlmgr install xcolor`
6. Install Rust: https://rust-lang.org/
7. Download and extract the Zenodo archive, then enter the extracted project root.
8. Compile MP-SPDZ and the 2PC malicious-secure protocols: `cd mp-spdz-0.4.2 && echo "MY_CFLAGS += -DINSECURE" » CONFIG.mine && make clean && make setup && make -j8 pairwise-offline.x mascot-offline.x lowgear-party.x mascot-party.x`. The `-DINSECURE` flag enables MP-SPDZ's insecure benchmarking functionality for local key generation; it is required for the artifact benchmarks (MP-SPDZ will otherwise report `You are trying to use insecure benchmarking functionality for local key generation`) and should not be used for production deployments. If you add the flag after a previous MP-SPDZ build, run `make clean` again before recompiling.

A.2.5 Benchmarks

We provide benchmarks for the following tasks.

1. Computation time and communication costs comparison for shuffling input sizes $2^{10}, 2^{12}, 2^{14}, 2^{16}$ on a single smaller local computer (Option 3) and input sizes $2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}$ on two AWS machines (Option 1) or two comparable large local machines/containers (Option 2). The supported shuffling benchmarks are FLOSS, PermNet, SimpleNet, and OPM.

2. Computation time and communication costs comparison for sorting 32-bit inputs with sizes $2^9, 2^{10}$ on a single smaller local computer (Option 3) and input sizes $2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}$ on two AWS machines (Option 1) or two comparable large local machines/containers (Option 2). The supported sorting benchmarks are Radix[FLOSS], Radix[PermNet], Radix[SimpleNet], StS[FLOSS], StS[PermNet], StS[SimpleNet], StS[OPM], and SortNet.

3. Offline monetary cost vs. online time single-point plot summary for shuffling 2^{20} inputs and sorting 2^{13} inputs, which requires the full-size data collected on two AWS machines (Option 1) or two comparable large local machines/containers (Option 2).

A.2.6 Limitations

The full paper plots require either two AWS `c5.18xlarge` instances (Option 1) or comparable local hardware (Option 2, two 72-vCPU/144GB servers or a single 144-vCPU/288GB server split into two Docker containers), and approximately 24 hours of compute time. The Option 3 single-computer workflow is intended as a functional and partial-results check: it omits the largest shuffle size 2^{20} and sorting size 2^{13} to avoid exceeding memory on a 16GB machine. Consequently, Option 3 cannot reproduce the offline monetary cost vs. online time Pareto plots; reviewers who wish to reproduce those plots should use the AWS workflow (Option 1) or the large local server workflow (Option 2). Reviewers using the single-server Docker variant of Option 2 should also be aware that both simulated parties share the same physical CPU, memory, and disk, so results may differ somewhat from a genuine two-machine deployment (Option 1 or the two-server variant of Option 2); we mitigate this by pinning each container to a disjoint set of CPU cores and memory and by rate-limiting the virtual network to 25 Gbps to match the AWS network bandwidth.

A.3 Set-up

In this section, we provide information about setting up and running the artifacts.

A.3.1 Installation

All setup starts by downloading and extracting the Zenodo archive.

Option 1 (AWS).

1. Install TeX Live/MacTeX, Python 3, the Python packages `numpy`, `matplotlib`, `pandas`, `scipy`, and `latex`, and AWS CLI version 2 on the personal host machine.

2. Configure AWS with `aws configure`, using `us-west-2` as the default region and `json` as the default output format.

3. Start the two-machine cluster from the project root with `python3 artifacts/start_cluster.py`.

Option 2 (large local server).

1. From the project root, build the provided Docker image, which installs all dependencies and builds FLOSS inside the container: `docker build -memory=8g -memory-swap=10g -t floss-large-local -f artifacts/local-large.Dockerfile .`

2. Create a Docker bridge network for the two simulated parties: `docker network create -driver bridge -subnet 172.28.0.0/16 floss-bench-net`.

3. Start one container per party: `docker run -dit -name floss-party0 -cpuset-cpus="0-71" -memory="144g" -shm-size="16g" -cap-add=NET_ADMIN -privileged=true -network floss-bench-net -ip 172.28.0.2 floss-large-local bash`, and similarly `docker run -dit -name floss-party1 -cpuset-cpus="72-143" -memory="144g" ... -ip 172.28.0.3 floss-large-local bash`.

4. Restrict each container's network to 25 Gbps to match the AWS setting: `docker exec floss-party0 bash -lc "tc qdisc replace dev eth0 root tbf rate 25gbit burst 32mb latency 50ms"` (and likewise for `floss-party1`).

5. Create the `parties.txt` config file inside both containers for the Rust benchmarks, e.g. `docker exec floss-party0 bash -lc "cd /root/repo && printf '172.28.0.2:8644\n172.28.0.3:8645\n' > parties.txt"` (repeated for `floss-party1`).

6. (Alternative) If instead using two separate physical local servers, install FLOSS and its dependencies natively on each server as in Option 3, and create the same `parties.txt` on both using an existing local network bridge between the two servers.

Option 3 (single local computer).

1. Install the OS libraries, Python, TeX, and Rust dependencies listed in the Software Dependencies section.

2. Compile MP-SPDZ from the project root with `cd mp-spdz-0.4.2 && echo "MY_CFLAGS += -DINSECURE" » CONFIG.mine && make clean && make setup && make -j8 pairwise-offline.x mascot-offline.x lowgear-party.x`

`mascot-party.x` (the `-DINSECURE` flag enables the insecure local key generation needed by the benchmarks; see the Software Dependencies section for details).

A.3.2 Basic Test

After setup, one can quickly check the artifact's functionality by running one small FLOSS benchmark test.

Option 1 (AWS).

1. Run `python3 artifacts/start_cluster.py`; this creates the two EC2 instances and copies the required configuration files, including `parties.txt`, to both machines.
2. Run `python3 artifacts/run_floss_bench.py`; a few synchronous tests will complete in roughly 5–10 minutes before the long-running benchmarks are launched in the background.

Option 2 (large local server).

1. After the two containers are running, open two terminal tabs, one for each party.
2. On party 0, run `docker exec -it floss-party0 bash -c "cd /root/repo && env ALONE=false RANK=0 IP_FILE=parties.txt cargo bench -bench floss_shuffle"`.
3. At approximately the same time, on party 1, run the same command with `RANK=1` on `floss-party1`.

Option 3 (single local computer).

1. Run `cargo bench -bench floss_shuffle`; completion produces FLOSS shuffle times and confirms that the Rust and MP-SPDZ components are usable.

For all options, verify beforehand that `parties.txt` has been created correctly in the project root on *both* parties (each entry gives that party's network address and port)–an incorrect or missing `parties.txt` leads to connection failures between the two parties.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): FLOSS achieves an online computation time speedup of 800× and communication cost speedup of 6× over state-of-the-art alternatives for 2PC shuffling over state-of-the-art alternatives. This is proven by experiment (E1). The result is described in Section 7.2 and is illustrated in Table 1, Figure 7, and Figure 8.

(C2): FLOSS achieves an offline monetary cost reduction of 2× for 2PC shuffling over state-of-the-art alternatives. This is proven by experiment (E1). The result is described in Section 7.3 and is illustrated in Figure 7.

(C3): FLOSS requires ≈ 2× less SPDZ preprocessing than PermNet for 2PC shuffling. This is proven by experiment (E1). The result is described in Section 7.3 and is illustrated in Table 3.

(C4): Radix[FLOSS] achieves an online computation time speedup of 5× and communication cost speedup of 3× for sorting over state-of-the-art alternatives. This is proven by experiment (E2). The result is described in Section 7.4 and is illustrated in Figure 9 and 10.

(C5): Radix[FLOSS] achieves an offline monetary cost reduction of 3× for sorting over state-of-the-art alternatives. This is proven by experiment (E2). The result is described in Section 7.4 and is illustrated in Figure 9.

A.4.2 Experiments

(E1): [Shuffling baseline comparisons] [1 human-hour + 24 compute-hours (running full tests on AWS) or 3 compute-hours (running subsampled tests locally) + 256GB disk]: This experiment will describe the offline and online costs for the shuffling baselines. It will automatically run the shuffling (and sorting) benchmarks, and they will automatically collect both the offline/online times and communication cost into individual CSV files. This experiment will specify the plotting scripts and output files needed to validate the main *shuffling* claims from the paper.

Preparation: Complete the AWS (Option 1), large local server (Option 2), or single-computer (Option 3) setup from Section A.3. For the full AWS run, configure AWS credentials on the host machine and run `python3 artifacts/start_cluster.py` from the project root. For the large local server run, build `artifacts/local-large.Dockerfile`, start the two party containers, and create `parties.txt` on both, as described in Section A.3. For the local subsampled run, compile MP-SPDZ and ensure Docker is available for the OPM baseline.

Execution: For the full AWS run, execute the benchmark suite with `python3 artifacts/run_floss_bench.py`. The initial setup takes approximately 5–10 minutes and may require accepting prompts with `Y`; after that, the benchmarks run asynchronously under `nohup`. Wait approximately 24 hours, then collect results with `python3 artifacts/collect_floss_bench_results.py`. After collecting results, tear down the cluster with `python3 artifacts/teardown_cluster.py`.

For the large local server run, open one terminal per party and, at approximately the same time, run on

party 0 (e.g. `docker exec -it floss-party0 bash`, then `cd /root/repo`): `env ALONE=false RANK=0 IP_FILE=parties.txt cargo bench -bench <name>` for each of `simple_perm_network_shuffle`, `perm_network_shuffle`, `floss_shuffle`, `sort_with_simple_perm_network`, `sort_with_floss`, `sort_with_perm_network`, `sort_with_quicksort`, and `sort_with_sorting_network`, followed by `./scripts/bench_opmcc.sh 0 0 172.28.0.2:39530,172.28.0.3:39531`; run the identical commands with `RANK=1` on `floss-party1`, with the OPM script argument `./scripts/bench_opmcc.sh 0 1 172.28.0.2:39530,172.28.0.3:39531` (substitute your containers'/servers' actual IPs if different). If any individual benchmark fails partway, it can be re-run in isolation. This takes approximately 24 compute-hours, comparable to Option 1. Afterwards, collect the output CSVs from party 0 onto the host: `docker exec floss-party0 bash -lc "mkdir -p /root/results && cp /root/repo/*.csv /root/results/"` then `docker cp floss-party0:/root/results/. .` from the project root.

For the local subsampled run, build and run the OPM baseline in Docker with `cd OPM/`, `docker build -t mosac:latest .`, `docker run -it -name mosac-dev -cap-add=NET_ADMIN -privileged=true mosac:latest bash`, and then `./scripts/bench_opmcc.sh 1` inside the container. Exit Docker and copy `shuffle_opmcc_offline.csv` and `shuffle_opmcc_online.csv` from the container path `mosac-dev:/opm/` to the project root using `docker cp`. Then run the remaining local benchmarks: `cargo bench -bench simple_perm_network_shuffle`, `cargo bench -bench perm_network_shuffle`, `cargo bench -bench floss_shuffle`, `cargo bench -bench sort_with_simple_perm_network`, `cargo bench -bench sort_with_floss`, `cargo bench -bench sort_with_perm_network`, `cargo bench -bench sort_with_quicksort`, and `cargo bench -bench sort_with_sorting_network`.

Results: Generate plot data from the project root by running `cd plots/`, `python3 gen_plot_data.py`, `pdflatex main.tex`, and `cd ..` for the AWS (Option 1) or large local server (Option 2) workflow; this can be run on the host after installing the plotting dependencies, or inside the party-0 container/server. For the local subsampled workflow (Option 3), use `python3 gen_plot_data.py -alone 1` instead. The expected shuffling outputs are `plots/plot_data/shuffle_offline_time.csv`, `plots/plot_data/shuffle_online_time.csv`, `plots/main.pdf`, `shuffle_floss_offline.csv`, and `shuffle_perm_network_offline.csv`. To interpret the results with the main claims:

- `(Experiment E1) Table 1 + Individual shuffling data` provides the online and offline computation and communication detailed costs for all the shuffling baselines for Claim C1. FLOSS is expected to have the fastest online computation time and lowest online communication cost over all shuffling baselines. There is an approximate online time speedup of 800× and communication cost speedup of 6×.
- `(Experiment E1 + E2) Figures 7-10` provides the offline/online cost shuffling trend plots for Claim C1 and the offline monetary cost vs. online time shuffling plot for Claim C2. FLOSS has 2× lower offline monetary cost than all other baselines.
- `(Experiment E1) Table 3 (in Project Root directory)` provides the offline preprocessing breakdown costs for Claim C3. FLOSS has $\approx 2\times$ less SPDZ preprocessing than PermNet for 2PC shuffling.

(E2): [Sorting baseline comparisons] [1 human-hour + 24 compute-hours (running full tests on AWS) or 3 compute-hours (running subsampled tests locally) + 256GB disk]: This experiment will describe the offline and online costs for the sorting baselines. It will automatically run the sorting (and shuffling) benchmarks, and they will automatically collect both the offline/online times and communication cost into individual CSV files. This experiment will specify the plotting scripts and output files needed to validate the main *sorting* claims from the paper. **Note that if you already ran the "Preparation" and "Execution" stages in Experiment E1, do not re-run either of these stages for Experiment E2. Only run the "Results" stage since all results are collected already.**

Preparation: Use the same preparation as in Experiment E1. If E1 has already been prepared and executed, do not repeat preparation or execution–the E1 workflow collects the sorting data needed for E2 as well.

Execution: Use the same execution as in Experiment E1 **if not performed Experiment E1 execution yet.** For the full AWS run, execute `python3 artifacts/run_floss_bench.py`, wait approximately 24 hours, and collect results with `python3 artifacts/collect_floss_bench_results.py`; tear down the cluster afterwards with `python3 artifacts/teardown_cluster.py`. For the large local server run, run the same per-party `cargo bench` and `bench_opmcc.sh` commands listed under E1 on both parties simultaneously, then collect the CSVs from party 0 as in E1. For the local subsampled run, run the Docker-based OPM commands and the local

`cargo bench` commands listed under E1. The sorting benchmarks are `sort_with_simple_perm_network`, `sort_with_floss`, `sort_with_perm_network`, `sort_with_quicksort`, and `sort_with_sorting_network`.

Results: Generate plot data as in E1: `cd plots/`, then `python3 gen_plot_data.py` for AWS or large local server results, or `python3 gen_plot_data.py -alone 1` for local subsampled results, followed by `pdflatex main.tex`. The expected sorting outputs are `plots/plot_data/sort_offline_time.csv`, `plots/plot_data/sort_online_time.csv`, and `plots/main.pdf`. To interpret the results with the main claims:

- `(Experiment E2) Individual sorting data` provides the online and offline computation and communication detailed costs for all the sorting baselines for Claim C4. Radix[FLOSS] is expected to have the fastest online computation time and lowest online communication cost over all sorting baselines. There is an approximate online time speedup of 5× and communication cost speedup of 3×.
- `(Experiment E1 + E2) Figures 7-10` provides the offline/online cost sorting trend plots for Claim C4 and offline monetary cost vs. online time sorting plot for Claim C5. Radix[FLOSS] has 3× lower offline monetary cost than all other baselines.

A.5 Notes on Reusability

FLOSS is well suited for performing privacy-preserving computations in the malicious-secure 2PC setting. Past malicious-secure shuffling protocols have been well studied in the malicious-secure 3PC honest-majority setting; however, this requires setting up an additional honest third party beyond the two-party case.

FLOSS is flexible, such that any future developer who wishes to add their own privacy-preserving protocol can do so easily within the framework. The implementation is extensible, allowing developers to define additional higher-level custom gates on top of the gates we provide, making it easy to reuse and construct more complex interactive arithmetic permutation circuits. For instance, the sorting case study demonstrates the power of using FLOSS's arithmetic permutation circuit framework.

An exciting future direction is to implement more complicated privacy-preserving analytics protocols through FLOSS, such as database joins and heavy hitters.

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/.