

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Remise: Authorized Anonymous Communication Systems

Rohan Ravi, Paritosh Shukla, and Adithya Vadapalli, *IIT Kanpur*

https://www.usenix.org/conference/usenixsecurity26/presentation/ravi-rohan

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: Remise: Authorized Anonymous Communication Systems

Rohan Ravi, Partiosh Shukla, and Adithya Vadapalli

A Artifact Appendix

A.1 Abstract

This artifact provides the C++ prototype and benchmarking harness for REMISE, a two-server authorized anonymous communication system built on Distributed Oblivious RAM. It contains the two server-side benchmark binaries evaluated in the paper—`remise` (the REMISEBB bulletin-board variant) and `remise_sabre` (the REMISEBB (SABRE) variant whose write-authorization uses an MPC evaluation of a bitsliced LowMC PRF)—together with the DPF/FSS primitives, secret-sharing and MPC building blocks, and the coroutine-based asynchronous networking framework used for protocol execution. The harness runs the two servers as separate processes that communicate over a TCP socket, emulates wide-area network conditions with `tc netem` (latency) and `tbf` (bandwidth), and reports per-request audit latency, online validity-check latency, end-to-end throughput, and authorized-request goodput, appending each run to a CSV. We package everything as a two-container Docker setup (servers `p0` and `p1`) so that an evaluator can reproduce the request-validity-check and throughput/goodput trends with a small number of commands. The artifact reproduces the qualitative and quantitative trends of Figures 3 and 4 of the paper, including the order-of-magnitude online validity-check speedup of REMISEBB at large database sizes.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact runs entirely on the evaluator's own machine using *synthetic* data: databases, indices, messages, and authorization tokens are generated internally, and no real user data is collected, transmitted, or stored. There are no destructive operations on the host. Two points warrant the evaluator's attention:

- **Elevated container capabilities for network emulation.** To apply `tc netem`/`tbf` shaping *inside* the experiment containers, the two server containers are granted the Linux `NET_ADMIN` capability and are run with `apparmor=unconfined`. These settings are scoped to the two experiment containers on a private user-defined Docker bridge network; they do not alter the host's networking or security configuration. An evaluator who prefers not to grant these can run all experiments at zero emulated latency (LAN conditions) and still validate the computational trends—only the latency-dependent rows of Figure 3 require shaping.
- **Teardown.** The provided teardown script stops and removes only the two experiment containers and (optionally) the built image; it touches no other host state.

A.2.2 How to access

The artifact is archived on Zenodo at https://zenodo.org/records/20438384. Download and extract the archive; all paths below are relative to its root.

However, for the evaluation, we recommend cloning the repo from https://github.com/anonymoususer99946996/anonymous.git and then going into the Remise folder.

A.2.3 Hardware dependencies

- **CPU.** An x86-64 processor with the `AES-NI`, `SSE2`, `SSSE3`, and `AVX2` instruction-set extensions. `AES-NI` is used by the DPF pseudorandom generator; `AVX2` (256-bit vectors) is used by the bitsliced LowMC audit in REMISEBB (SABRE). The build uses `-march=native`, so the evaluation host should be the run host.
- **Memory.** Memory scales with database size and message size. The paper's measurements up to 2^{30} entries were taken on a machine with 251 GiB of RAM. The headline claims (database size 2^{26}) are reproducible with substantially less; we recommend at least $\approx$ 16–32 GiB to reach 2^{26} comfortably.
- **Cores / hosts.** The two servers may run as two containers on a single multi-core machine (the default), or on two separate machines on the same network. Two physical cores suffice; more cores do not change the reported per-request metrics.

The paper's reference platform was a server with two Intel® Xeon® Gold 6430 processors, 251 GiB of RAM, running 64-bit Linux.

A.2.4 Software dependencies

The recommended path is Docker (a single image builds both binaries). The provided `Dockerfile` is based on `ubuntu:22.04` and installs all dependencies; the only host requirement is a working Docker installation (Docker Engine with the `compose` plugin) on a Linux host.

For a native (non-Docker) build, the dependencies are: Ubuntu 22.04 (or equivalent), `g++-12` with C++20 support (required for coroutines), Boost 1.81 headers (the asynchronous networking layer uses `boost::asio` coroutines and `experimental/awaitable_operators`, which are not present in Ubuntu's default Boost 1.74), `libbsd` (`arc4random`), OpenSSL/`libcrypto` (used by the LowMC audit streams), `iproute2` (`tc`, for latency/bandwidth emulation), and `make`. The Boost 1.81 headers are vendored into the image at build time, so the Docker path needs no separate Boost installation.

A.2.5 Benchmarks

None. All workloads (databases, write/read requests, authorized vs. cover traffic, authorization tokens) are generated internally and parameterized on the command line. No external data sets or models are required.

A.3 Set-up

A.3.1 Installation

1. Install Docker Engine with the `compose` plugin on a Linux host.
2. From the artifact root, build the image and start the two server containers:
   ```
   ./up.sh            # build & start
   ./up.sh --rebuild  # force rebuild
   ```
   The build compiles both benchmark binaries (`make remise` and `make remise_sabre`) and verifies that `/app/remise` and `/app/remise_sabre` exist in the image. The two containers (`p0`, the server/acceptor, and `p1`, the client) then run idle on a private bridge network, ready to receive experiments. Results are written to the host directory `./results/`, which is mounted into the containers.
3. Tear everything down when finished:
   ```
   ./down.sh          # stop & remove
   ./down.sh --rmi    # and remove image
   ```

Native build (optional). With the dependencies above installed and the Boost 1.81 headers available, run `make remise` and `make remise_sabre` (the Makefile honors `BOOST_ROOT=$HOME/boost-1.81`).

A.3.2 Basic Test

With the containers up, run a single small REMISEBB (SABRE) request at 10 ms RTT and 100 Mbit/s, a 2^{16}-entry database, 32-byte messages (leaf size 2), all requests authorized, 4 requests:

```
./run_remise_sabre.sh 10 100mbit 16 2 1.0 4
```

The script shapes the link, launches `p0` (server, backgrounded) and then `p1` (client, foreground), waits for completion, and clears the shaping. Expected output ends with an `Experiment Summary` block reporting (among others) the average audit latency, the average online latency (audit → finalize → FCW exchange → database update), bandwidth, throughput, and goodput, and appends a row to `./results/`. An analogous smoke test for REMISEBB uses `./run.sh 10 100mbit 16 2 1.0 4`. Successful completion of either (a printed summary and a new CSV row, no crash) confirms the build, the two-party networking, and the latency shaping are all functioning.

Execution. The experiments are driven by per-figure sweep scripts that loop over the database sizes, latencies, and authorized fractions of the corresponding figure and append one CSV row per trial; companion Python scripts read those CSVs and emit the plots. The underlying run scripts are:

`run.sh` one REMISEBB run:

```
./run.sh <RTT_ms> [bw] [log_nitems]
[leafsize] [auth_fraction] [num_requests]
[batch_size]
```

`run_remise_sabre.sh` one REMISEBB (SABRE) run:

```
./run_remise_sabre.sh <RTT_ms> [bw]
[log_nitems] [leafsize] [auth_fraction]
[num_requests]
```

where `log_nitems` is $\log_2$ of the database size, the message size in bytes is $16 \times$ `leafsize` (so `leafsize` $= 2$ gives the 32-byte messages used throughout), and `auth_fraction` is the *fraction of authorized requests*—i.e. the paper's "f% unauthorized traffic" corresponds to `auth_fraction` $= 1 - f/100$. The run scripts forward the requested RTT and a trial index to the binary (via the `REMISE_RTT_MS` and `REMISE_TRIAL` environment variables) so that each CSV row is self-labeled with its latency and repetition.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *The server-side request-validity-check (audit + authorization) time of* REMISEBB *is competitive at small*

database sizes and increasingly favorable at large ones; considering only the online phase, REMISEBB *attains an order-of-magnitude speedup over Spectrum (PACL), reaching nearly* $80\times$ *at database size* 2^{26}. This is evaluated by experiment (E1) and corresponds to Figure 3a and the discussion in Section 5 ("Request Validity Check Comparisons").

(C2): *The request-validity-check time of* REMISEBB (SABRE) *is essentially independent of the database size, because its PRF-based authorization is* $O(1)$ *and its audit is* $O(\log n)$*; consequently it is increasingly favorable relative to Express (PACL), whose DPF-based authorization grows linearly with the database size.* This is evaluated by experiment (E2) and corresponds to Figure 3b.

(C3): *For both* REMISEBB *and* REMISEBB (SABRE)*, throughput and goodput behave as reported in Figure 4: with offline-preprocessed DPFs, increasing the fraction of unauthorized traffic* improves *throughput (unauthorized requests terminate after the lightweight authorization phase and skip the expensive access phase), whereas generating DPFs on-the-fly reduces both; and* REMISEBB (SABRE) *with on-the-fly DPFs surpasses Express (PACL) at progressively smaller database sizes as the unauthorized fraction grows.* This is evaluated by experiment (E3) and corresponds to Figures 4a and 4b.

(C4): *For both* REMISEBB *and* REMISEBB (SABRE)*, the preprocessing (DPF interval evaluation) time and the audit time are independent of the message size, whereas the access time (finalize, FCW exchange, and database update) grows linearly with the message size.* This is evaluated by experiment (E4). This claim corresponds to Figure 5 from our paper.

Scope note on baselines. Reproducing the Spectrum (PACL) and Express (PACL) curves requires the PACL codebase (https://github.com/sachaservan/pacl; they are also available in the `pacl` folder in our artifact); they are currently not part of this artifact. For the *Functional* and *Results Reproduced* badges, the experiments below focus on reproducing the REMISE curves of Figures 3 and 4 (the contribution of this paper); the cross-system speedup factors in C1–C3 are then obtained by comparing against the published PACL measurements (or against a local PACL build, if the evaluator chooses to set one up).

A.4.2 Experiments

Timing conventions used below. Each request proceeds through *eval* (the `__evalinterval_mpc` DPF interval evaluation), *audit* (the validity check—an XOR-tag exchange for REMISEBB, a bitsliced-LowMC MPC for REMISEBB (SABRE)), and *write* (finalize, the FCW exchange, and the database update). DPF *generation* is offline and is never inside any timed region. We report two timings that differ only in whether *eval* is counted:

- **online** = audit + finalize + FCW exchange + DB update (eval *excluded*); this is the "with preprocessed DPFs" setting.
- **total** (a.k.a. "on-the-fly") = eval + audit + write (eval *included*, generation still excluded). This called "on-the-fly", because here we assume that the DPFs are produced as and when we need it.

Both numbers are produced by a *single* run—no separate mode flag is needed—because they are simply the same work measured against two different denominators.

(E1): *RemiseBB validity-check time (Figure 3a)* [15 human-minutes + ≈2–4 compute-hours + a few GiB to ∼tens of GiB RAM depending on the largest database size]: measure the per-request validity-check time of REMISEBB as a function of database size, at three latencies, reporting both the *online* time (audit only, eval excluded) and the *total* time (eval + audit), underlying C1.

Preparation: Bring the containers up (`./up.sh`). The sweep uses 32-byte messages (`leafsize` = 2), all-authorized traffic (`auth_fraction` = 1.0), exactly one request per run (`num_requests` = 1, `batch_size` = 1), and averages over 30 trials per point. The default database range is 2^{16} through 2^{26} at latencies 10/30/60 ms; pass arguments to cap the size or trial count (e.g. `./sweep_fig3a.sh 18 3` for a quick check).

Execution: Collect the data and produce the plots:

```
./sweep_fig3a.sh
python3 plot_fig3a.py --combined
```

Results: `sweep_fig3a.sh` writes one row per trial to `results/fig3a_remisebb.csv` with columns `variant,latency_ms,log_nitems,trial,online_ms,total_ms`, where `online_ms` is the audit-only time and `total_ms` is eval+audit. `plot_fig3a.py` averages over the 30 trials (with 95% confidence intervals) and emits one plot per latency (10/30/60 ms) plus a combined panel, each showing the *online* and *total* curves vs. database size. Compare against Figure 3a. To obtain the C1 speedup factor, divide the Spectrum (PACL) online time (from the paper, or a local PACL run) by the REMISEBB `online_ms` at the same database size; at 2^{26} this ratio should be close to $80\times$.

(E2): *RemiseBB (Sabre) audit time (Figure 3b)* [15 human-minutes + ≈1–2 compute-hours]: measure the audit (validity-check) time of REMISEBB (SABRE) as a function of database size and confirm it is (near-)constant in the database size, underlying C2. The Sabre audit is the bitsliced-LowMC MPC (`encrypt2_p0p1`); there is no preprocessing split, so a single *audit* time is reported.

Preparation: Same workload as E1 (`leafsize` = 2, `auth_fraction` = 1.0, one request per run, 30 trials, 2^{16} through 2^{26} at 10/30/60 ms), but driven through `remise_sabre`.

Execution: `./sweep_fig3b.sh`
`python3 plot_fig3b.py --combined`
Results: `sweep_fig3b.sh` writes one row per trial to `results/fig3b_remisebb_sabre.csv` with columns `variant,latency_ms,log_nitems,trial,audit_ms`. `plot_fig3b.py` averages over trials and plots audit time vs. database size for each latency; confirm the curve stays essentially flat as the database grows (the audit cost is independent of the database size), in contrast to Express (PACL) whose authorization grows linearly. Compare against Figure 3b.

(E3): *Throughput and goodput, with-preproc and on-the-fly DPFs (Figures 4a and 4b)* [20 human-minutes + ≈2–3 compute-hours]: measure throughput and goodput while varying database size and the fraction of unauthorized traffic, for both REMISEBB (4a) and REMISEBB (SABRE) (4b), underlying C3.
Preparation: The emulated latency is fixed at 30 ms and the message size at 32 bytes (`leafsize` $= 2$). The figure's three panels correspond to 30%/60%/90% unauthorized traffic, i.e. `auth_fraction` $\in \{0.7, 0.4, 0.1\}$; the sweep iterates these automatically. Throughput is meaningful only with many requests per run, so the sweep uses a large `num_requests` (default 200) and several trials. *No DPF-mode flag is required:* each run reports both the with-preproc curves (*online* throughput/goodput, eval excluded) and the on-the-fly curves (*total* throughput/goodput, eval included).
Execution: Collect both variants and plot each figure:

```
./sweep_fig4.sh both
python3 plot_fig4.py \
    results/fig4a_remisebb.csv \
    --out fig4a \
    --title "RemiseBB"
python3 plot_fig4.py \
    results/fig4b_remisebb_sabre.csv \
    --out fig4b \
    --title "RemiseBB (Sabre)"
```

(Use `./sweep_fig4.sh a` or `b` to run a single variant, and a trailing `MAXK TRIALS REQS` to scale down for a quick check, e.g. `./sweep_fig4.sh a 18 2 50`.)
Results: `sweep_fig4.sh` writes one row per trial to `results/fig4a_remisebb.csv` and `results/fig4b_remisebb_sabre.csv` with columns `variant,unauth_frac,log_nitems,trial,throughput,goodput,online_throughput,online_goodput`. Each `plot_fig4.py` call averages over trials and renders the three-panel row (30%/60%/90% unauthorized), with four curves per panel: with-preproc throughput/goodput (the `online_*` columns) and on-the-fly throughput/goodput (the plain columns). Confirm the three qualitative effects of C3: (i) with preprocessed DPFs, higher unauthorized fraction raises throughput/goodput; (ii) on-the-fly DPFs lower both; and (iii) for REMISEBB (SABRE), the on-the-fly curves overtake Express (PACL) at smaller database sizes as the unauthorized fraction grows. Compare against Figures 4a and 4b.

(E4): *Message-size scaling of preprocess, audit, and access* [15 human-minutes + ≈1 compute-hour]: measure how the three per-request cost components scale with message size, for both REMISEBB and REMISEBB (SABRE), underlying C4. The three components are *preprocess* (the `__evalinterval_mpc` DPF interval evaluation), *audit* (the validity check), and *access* (finalize, FCW exchange, and database update).
Preparation: The database size is fixed at 2^{20} and the latency at 30 ms; the message size is swept by varying `leafsize` $\in \{2, 64, 256\}$ (i.e. 32, 1024, and 4096 bytes, since message size $= 16 \times$ `leafsize`). Each point uses one authorized request per run and averages over 30 trials. The sweep runs both binaries.
Execution: Collect both variants and plot each:

```
./sweep_msgsize.sh both
python3 plot_msgsize.py \
    results/msgsweep_remisebb.csv \
    --out msg_remisebb \
    --title "RemiseBB"
S=results/msgsweep_remisebb_sabre.csv
python3 plot_msgsize.py "$S" \
    --out msg_sabre \
    --title "RemiseBB (Sabre)"
```

Results: `sweep_msgsize.sh` writes one row per trial to `results/msgsweep_remisebb.csv` and `results/msgsweep_remisebb_sabre.csv` with columns `variant,latency_ms,leafsize,log_nitems,trial,preprocess_ms,audit_ms,access_ms`. Each `plot_msgsize.py` call averages over the 30 trials (with 95% confidence intervals), prints a summary table, and renders a stacked bar chart of preprocess/audit/access vs. message size. Confirm C4: the preprocess and audit segments stay essentially constant across the three message sizes, while the access segment grows roughly linearly with the message size.

In all experiments the per-trial CSV files (and the plots derived from them) are the primary artifacts of record; the network conditions are those reported in the paper (`tc netem` latency, bandwidth shaping via `tbf`). Expected outcomes are the curves of Figures 3 and 4; small deviations in absolute timing are expected across hardware, but the trends—and the order-of-magnitude online speedup underlying C1—should hold.

A.5 Notes on Reusability

The harness is parameterized so evaluators can explore beyond the reported points. `log_nitems` scales the database

size; `leafsize` scales the message size (bytes $= 16 \times$ `leafsize`); `auth_fraction` sweeps the authorized/cover-traffic mix; `num_requests` sets how many requests a run issues; and (for REMISEBB) `batch_size` controls how many requests are pipelined concurrently. Network conditions are independent of the binaries: `netshape.sh` applies and clears latency (`netem`) and rate (`tbf`) shaping inside the containers, splitting the requested RTT across the two endpoints, so arbitrary latency/bandwidth points can be evaluated without rebuilding. Because the two servers communicate only over a TCP socket and `p1` resolves its peer via the `P0_HOST` environment variable, the same binaries can be run across two physical machines (set `P0_HOST` to `p0`'s address and open port `9200`) to measure over a real network instead of an emulated one. The per-figure sweep scripts (`sweep_fig3a.sh`, `sweep_fig3b.sh`, `sweep_fig4.sh`) and their plotting companions (`plot_fig3a.py`, `plot_fig3b.py`, `plot_fig4.py`) are thin wrappers over this interface and are easy to adapt to new latency points, database sizes, or authorization mixes.

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/.