# Batched Differentially Private Information Retrieval

Kinan Dak Albab, *Brown University;* Rawane Issa and Mayank Varia, *Boston University;* Kalman Graffi, *Honda Research Institute Europe*

https://www.usenix.org/conference/usenixsecurity22/presentation/albab

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

## A Artifact Appendix

### A.1 Abstract

This artifact contains the C++ source code of our novel DP-PIR protocol that we introduce in the paper. Our protocol allows clients to privately query the contents of a remote database, without revealing information about their query to the service beyond a well-defined differentially private leakage. Unlike previous PIR protocols, DP-PIR amortizes queries from independent clients leading to constant amortized server and client side computation and communication complexity when the query volume is sufficiently large with respect to the size of the database.

Our artifact includes the code for the different entities of our protocol. This includes the client(s) code, as well as the two or more parties that constitute the service. In addition, the artifact includes scripts to run the various experiments and produce the plots and tables we show in the paper, including the comparisons with existing three existing protocols: Checklist, DPF, and SealPIR. Our implementation uses Google's Bazel build system, and includes Bazel ports for building the three aforementioned baselines. We developed our implementation using Ubuntu 20.04 and g++-11. We will provide a Docker container with all required dependencies by the artifact submission deadline.

The primary purpose if this artifact is to (1) support the claims of our paper about the efficacy of amortization with DP-PIR compared to current state of the art protocols, and (2) demonstrate how the performance of DP-PIR is governed by the different application parameters. To that end, we designed and ran several experiments that run our implementation or the baselines with different parameters and report the total service time taken to process the generated query loads. We ran our experiments on AWS instances to mimic a realistic setup where the different parties making up the protocol are deployed over separate machines and communicate over realistic networks. Most experiments can be run locally without AWS, except a couple of the larger data points that will most likely run out of memory when all the parties are run on the same local machine. We will provide detailed instructions on how to run experiments locally or over AWS, and how to interpret and plot the results by the deadline.

### A.2 Artifact check-list (meta-information)

- **Algorithm:** This artifact provides an implementation of DP-PIR, a new private information retrieval protocol.
- **Compilation:** We tested our artifact using G++-11. Our implementation uses Google's Bazel build system.
- **Binary:** No binaries are included. The protocol binary should be build from the source code using Bazel in optimized mode.
- **Run-time environment:** We developed our artifact on Ubuntu 20.04. We will provide a docker container with all the relevant dependencies.
- **Run-time state:** Our implementation, and especially the online portion of our protocol, is extremely sensitive to network bandwidth. In our experiments, we deployed our AWS instances in a cluster placement group to minimize network costs.
- **Execution:** Each data point on any of the plots in the paper is a separate running job spawned by our scripts. For the smaller data points, the execution takes a few seconds, but for the larger ones (e.g. 100M queries), it may take close to an hour.
- **Security, privacy, and ethical concerns:** There are no such concerns.
- **Metrics:** We report the total service side execution time. Concretely, this is the wall time between the first party/server in the protocol receiving the last query in the batch, right before processing of the batch starts, and the wall time at that same server right after the batch has been processed, and before the responses are sent to client(s). We report similar measurements for the baselines as well.
- **Output:** The experiments produce log files for the different parties, each file containing various debugging information as well as time measurements. Our artifact includes scripts that automatically process these files to extract the relevant information, and produce plots similar to the ones shown in the paper.
- **Experiments:** Our experiments are run via an "orchestrator" command line program provided in the artifact. This orchestrator is a simple nodejs web server that workers (local or AWS) ping for jobs. Aside from running these workers once, evaluators need only interact with the orchestrator via its control interface, e.g. they can use command 'load figure1' to direct the orchestrator to load and run the experiments needed to produce figure 1 in the paper.. The orchestrator is responsible for translating input commands into jobs, assigning them to workers, and tracking the progress of these workers including acquiring their output files.
- **How much time is needed to complete experiments:** In our setup, the experiments require about 14 hours of mostly *passive* running time to produce the results shown in the paper.
- **Publicly available:** at `https://github.com/multiparty/DP-PIR/tree/usenix2022`.
- **Code licenses:** MIT

### A.3 Description

#### A.3.1 How to access

Clone this repository `https://github.com/multiparty/DP-PIR/tree/usenix2022`.

#### A.3.2 Hardware dependencies

We ran our experiments using one r4.xlarge AWS instance per server/party. These instances have 2 vCPUs and 30.5GB RAM. If run locally, more memory will be required to run the larger experiments, since all the parties (and thus all their memory) will be run on the same machine. In such cases, we recommend that proportionally smaller experiments are run to fit the hardware constraints. The artifact documentation includes more details on this.

## A.4 Installation

We provide a Docker container that includes all require dependencies. Instructions on building and running this container are provided in the artifact. We also provide instructions on how to deploy and run the experiments locally or via AWS.

## A.5 Experiment workflow

To simplify running experiments, we provide an orchestrator program included in the artifact. The orchestrator takes care of configuring the protocol per the experiment parameters. The orchestrator is ideal for experiments with many parties or parallel machines, as it automatically assigns the experiment tasks to the workers and monitors their progress.

At a high level, our workflow with the orchestrator follows these steps:

1. The orchestrator application is run via the command line.
2. Several workers are spawned, either as AWS instances or locally via the provided scripts. As many workers are needed as the sum of parties and clients. For most experiments in the paper, this translates to 3 workers needed for 2 parties and 1 client.
3. The workers execute background daemon scripts that periodically ping the orchestrator to request jobs or report progress.
4. Evaluators issue commands to the orchestrator to run instances of our protocol with specific parameters. All the parameters for all of the results in the paper are packaged inside the artifact and can be loaded by name (e.g. 'load figure1'). However, evaluators can also run experiments with different parameters, which they need to specify to the orchestrator via an interactive dialog.
5. The workers receive the jobs corresponding to the different parameters issued by the evaluator. Workers run these jobs, which include running various steps of the protocols, such as creating queries, shuffling queries, and exchanging various messages over the network. Whenever a worker finishes a job, it reports that to the orchestrator along with the output file, which include the measured computation time.
6. The orchestrator notifies evaluators whenever workers and experiments are completed. The evaluators can then run the plotting script provided in the artifact to plot the results similar to the plots in our paper.

It is possible to run experiments directly using the protocol implementation without relying on the orchestrator. Consult the artifact documentation for more details.

## A.6 Evaluation and expected results

The main goal of this artifact is to produce plots showing the performance of DP-PIR as a function of the different application and setup parameters. Specifically, we are interested in demonstrating (1) how the performance of DP-PIR compares to that of existing protocols, and (2) how the performance of DP-PIR scales with different parameters.

With our protocol, we have the following parameters:

1. The database size: the number of rows in the database being privately queried. In our figures, this size ranges between 2.5 million rows for the larger experiments and 100K or 10K rows for the smaller ones.
2. The number of queries: how many queries to process via the protocol. This can range between several thousands and hundreds of millions. Note: on setups with limited RAM, the artifact will not be able to handle the larger query numbers as it will run out of memory.
3. The number of parties: how many parties constitute the service. Our protocol requires at least two parties and tolerates up to $n-1$ malicious and colluding parties. This is almost always set to 2 in our experiments, except figure 5 where it ranges between 2 and 5.
4. Parallelism: how many instances/workers/servers does a party possess. The more servers here the faster the protocol will run as the queries get split among these servers. We almost always set this to 1 except in table 2.
5. $\varepsilon$ and $\delta$: the differential privacy parameters governing how much leakage is tolerated. Smaller parameters imply more privacy at the cost of performance.
6. The mode: whether we are measuring the offline or online portions of our protocol.

When creating a job, the orchestrator will interactively request these parameters from the evaluators. Alternatively, the orchestrator can be instructed to load one or more bundled experiment which specifies all these parameters in accordance to the paper.

The main expected result here is a confirmation of the efficacy of our protocol and its amortization. Specifically, that our protocol becomes significantly faster than existing systems as the number of queries approaches or exceeds the database size. This is demonstrated by producing a plot similar to figure 1 in the paper: all the parameters are fixed (in the paper: DB size = 2.5M, parties = 2, parallelism = 1, $\varepsilon = 0.1$, $\delta = 10^{-6}$), while varying the number of queries (e.g. from $10^4$ to $10^8$). For each number of queries, we run both our *online* protocol and an existing PIR protocol (e.g. Checklist), and plot the reported runtimes as a function of the number of queries. We shows our results from the paper for demonstration below.

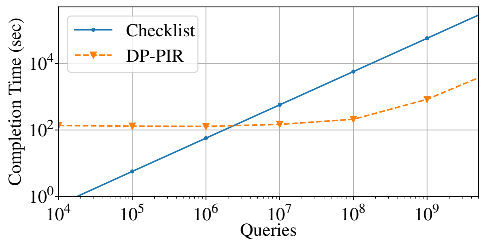

Figure 1: Checklist and DP-PIR Total completion time (y-axis, logscale) for varying number of queries (x-axis, logscale) against a 2.5M database

The exact numbers shown in the plot are setup dependent, and may significantly vary between setups. Our protocol is extremely susceptible to any changes in network bandwidth and latency. However, we expect to see three trends: (1) The total runtime of checklist is proportional to $O(|queries| \times \sqrt{|DB\ size|}$. (2) DP-PIR runtime initially is constant and does not seem to grow much with the number of queries. As the number of queries becomes similar in magnitude to the database size, our performance starts to grow with the number of queries. (3) Our protocol is (much) slower than Checklist for few queries, and much faster than Checklist for huge number of queries. Checklist's graph crosses over DP-PIR's somewhere in the middle, for a number of queries roughly in $O(|DB\ size|)$. A reasonable number of queries would be between 0.5 to 2.5 times the database size, depending on the setup.

If these three trends are observed, then the result match our expectations and confirms our claims about the performance of DP-PIR and its amortization. If either of them is absent, specifically, if DP-PIR remains slower than or comparative to Checklist even as the number of queries becomes large, that would disprove our performance and efficiency claims.

Another expected result is to demonstrate that DP-PIR scales with the different parameters as expected. Specifically, that it scales linearly in the number of queries and database size, for both online and offline stages, scales super-linearly in the number of parties in the offline stage, and exhibits close to linear speedups when additional parallel resources are used. These can be validated by fixing all the parameters except the parameter under investigation, and plotting the performance of DP-PIR as a function of that singular parameter. The produced plots should exhibit similar trends to the plots in section 7 of the paper.

## A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.