USENIX Security '24 Artifact Appendix: YPIR: High-Throughput Single-Server PIR with Silent Preprocessing

Samir Jordan Menon
Blyss

David J. Wu
UT Austin

A Artifact Appendix

A.1 Abstract

We implement YPIR (from Section 3) and YPIR+SP (from Section 4.3) in a single open-source Rust library. We provide a benchmarking binary that runs a YPIR client and server, and outputs the server runtime and the query and response sizes.

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical Concerns

Our implementation is not production-ready.

A.2.2 How to Access

The implementation is available at https://github.com/menonsamir/ypir/tree/b980152. The permanent DOI is https://doi.org/10.5281/zenodo.13117988.

A.2.3 Hardware Dependencies

The recommended hardware platform is an AWS `r6i.16xlarge` machine. The implementation should run well on any machine with a modern server CPU that supports AVX-512. The implementation will run with reduced performance on machines without AVX-512 support[1]. The YPIR experiments described here require 64 GB of memory and the HintlessPIR experiments require 512 GB.

A.2.4 Software Dependencies

We provide a self-contained Docker image at https://ghcr.io/menonsamir/ypir:latest.[2] Building the code directly requires Rust and a C/C++ compiler. The `rust-toolchain.toml` file fixes the Rust compiler version to `nightly-2024-02-07`, and standard Rust installations will automatically download and use this compiler version. The code was tested on Ubuntu 22.04.

[1] Add `--features server` to `cargo build` commands to build without AVX-512.

[2] The permanent URL for the specific Docker image used for this evaluation is https://ghcr.io/menonsamir/ypir@sha256:cf41936974074679bc3f08b582a8417b65cd957aa5d716a1daca61fd2d365006.

A.2.5 Benchmarks

None.

A.3 Set-up

In Ubuntu 22.04, install Docker and GCC 11.

A.3.1 Installation

No installation step is required (beyond installing Docker).

A.3.2 Basic Test

The basic test can be run using the command: `sudo docker run --security-opt seccomp:unconfined --cpus=1 ghcr.io/menonsamir/ypir:latest 2147483648 1`. This downloads the Docker container and runs it. The arguments correspond to running YPIR on a 256 MB database with 2^{31} 1-bit entries. We provide more details about the basic test in evaluation.md.

A.4 Evaluation Workflow

A.4.1 Major Claims

We make a major claim for each key figure and table in Section 4 of our paper.

(C1): For retrieving a single bit from an 8 GB database, YPIR achieves roughly 50% higher throughput, similar (within 10%) query size, and over 100× reduction in response size compared to HintlessPIR. This is an outcome of experiment (E1), whose results are presented in Table 1.

(C2): For retrieving a single bit from a 32 GB database, YPIR has slightly (within 5%) lower throughput, 50% larger query size, and the same response size as DoublePIR* (our DoublePIR implementation). Moreover, YPIR spends less than 5% of its server time on the LWE-to-RLWE translation. This is an outcome of experiment (E2), whose results are presented in Fig. 2 and Table 2.[3]

[3] The query and response sizes for DoublePIR* are included in Table 8 of the full version of this paper.

(C3): When considering cross-client batching with 4 clients and a 32 GB database, YPIR achieves up to a 40% increase in effective throughput. This is an outcome of experiment (E3), whose results are presented in Fig. 3.

(C4): For retrieving a 32-KB record from an 8 GB database, YPIR+SP achieves similar (within 10%) server throughput, similar (within 10%) query size, and over 5× reduction in response size compared to HintlessPIR. This is an outcome of experiment (E4), whose results are presented in Table 5.

A.4.2 Experiments

In the following commands, we use `$YPIR` to refer to the command prefix `sudo docker run --security-opt seccomp:unconfined --cpus=1 -it ghcr.io/menonsamir/ypir:latest`. Similarly, we use `$HINTLESSPIR` to refer to `sudo docker run --security-opt seccomp:unconfined --cpus=1 -it ghcr.io/menonsamir/hintlesspir:latest`. We measure the server computation time of YPIR averaged over 5 trials.[4]

(E1): [15 human-minutes + 2 compute-hours]: Runs YPIR and HintlessPIR to retrieve a bit from an 8 GB database.
Preparation: Run `sudo docker image pull ghcr.io/menonsamir/hintlesspir` to pre-download a Docker image for HintlessPIR we have created. The build specification for this container is available in hintlesspir-patch.patch.
Execution: Run YPIR with `$YPIR 68719476736 1`, and save the final "Measurement" output (15 minutes). Then, run HintlessPIR with `$HINTLESSPIR 8GB` (1.5 hours). Note the final value under "Time" in nanoseconds, and confirm that "Iterations" is 1.
Results: The throughput of each scheme is the database size in GB (8 in this case) divided by the server computation time in seconds. The server computation time for YPIR is in milliseconds in the measurement data in `online` under `server_time_ms`. Ensure you are using the `server_time_ms` value from the *online* section of the measurement data. The query and response sizes for YPIR are also in the `online` section, under `upload_bytes` and `download_bytes`, respectively. The HintlessPIR server computation time is reported under "Time", in nanoseconds. The HintlessPIR implementation does not output query and response sizes by default, so we use Lemma 7 from the HintlessPIR paper to calculate these in hintlesspir-notes.md.
Expected: YPIR should have a throughput of about 11 GB/s, and HintlessPIR should have a throughput of about 4.9 GB/s. Query and response sizes should match Table 1.

[4] Because HintlessPIR runs take longer (generally hours), we only run a single trial for these experiments. This does not appear to impact measurements significantly.

(E2): [30 human-minutes + 1 compute-hour]: Runs YPIR and DoublePIR* to retrieve a bit from a 32 GB database.
Execution: Run `$YPIR 274877906944 1`.
Results: Compute the throughput and query and response size for YPIR as in (E1). For DoublePIR*, the server response time (in milliseconds) is the sum of `first_pass_time_ms` and `second_pass_time_ms` from the `online` section of the measurement. The query and response sizes for DoublePIR* are `doublepir_query_bytes` and `doublepir_resp_bytes`. The fraction of the YPIR server computation time for performing LWE-to-RLWE is `ring_packing_time_ms` divided by `server_time_ms`.
Expected: YPIR and DoublePIR* should have a throughput of about 12 GB/s. Query and response sizes should match Table 8 of the full version of this paper.

(E3): [15 human-minutes + 1 compute-hour]: Runs YPIR with cross-client batching across 4 clients.
Execution: Run `$YPIR 274877906944 1 4`.
Results: The effective throughput is the product of the number of clients (4) by the database size in GB (32 GB), divided by the server computation time in seconds.
Expected: YPIR with cross-client batching across 4 clients should achieve an effective throughput of roughly 16 GB/s.

(E4): [30 human-minutes + 1 compute-hour]: Runs YPIR+SP and HintlessPIR to retrieve a 32 KB record from a 8 GB database.
Execution: Run YPIR+SP with `$YPIR 131072 524288 --is-simplepir`. Then, run HintlessPIR with `$HINTLESSPIR 8GB`.
Results: Compute the throughput and query and response sizes of each scheme as in (E1).
Expected: YPIR+SP and HintlessPIR should have a throughput of roughly 4.9 GB/s, and query and response sizes should match Table 5.

A.5 Notes on Reusability

The YPIR implementation is a Rust crate that can be reused by an existing Rust project by running `cargo add --git "https://github.com/menonsamir/ypir.git"`. The implementation can also be compiled to WebAssembly and run directly in a webpage. A demo of YPIR for retrieval of breached passwords in in the `demo/` folder of the repository.

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/usenixsec2024/.