USENIX Security '24 Artifact Appendix: VeriSimplePIR

Leo de Castro
Massachusetts Institute of Technology

Keewoo Lee
Seoul National University

A Artifact Appendix

A.1 Abstract

This artifact is a C++ implementation of VeriSimplePIR as well as VLHE PIR and SimplePIR. The library is meant as a stand-alone implementation with minimal dependencies. All protocols are parametrized by a database defined by a number of entries N and an entry bitwidth d. Internal scripts automatically select secure parameters based on the desired machine word size (either 32 bits or 64 bits).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The code can be accessed at this github repo (artifact evaluation branch): https://github.com/leodec/VeriSimplePIR/tree/3643bb7cbaae02da98a195c4d004d4d083e3ab88.

A.2.3 Hardware dependencies

This library was tested on Ubuntu 20.04.6 LTS on a machine with an Intel i7 core and 32 GB of RAM. It should work on any machine with `clang++` and `OpenSSL`.

A.2.4 Software dependencies

Aside from the compiler and OpenSSL, there are no software dependencies for this library.

A.2.5 Benchmarks

None.

A.3 Setup

A.3.1 Installation

Full instructions to build the library are in the `README.md` file in the github repo. These instructions essentially consist of running `make` in the top directory. Further details on the internal parameters to achieve optimal performance are in this `README.md` file.

A.3.2 Basic Test

The executable files are located in the `bin/demo` directory. To execute a basic test, run `./bin/demo/test/pir_test` from the top directory. This will run the following four tests with a 1 MiB database:

1. Semi-honest PIR (SimplePIR).
2. VLHE PIR.
3. VeriSimplePIR without the proof generation. This is functionally equivalent to the first test, but it uses the VeriSimplePIR class.
4. Full VeriSimplePIR test.

A.4 Evaluation workflow

A.4.1 Major Claims

VeriSimplePIR Performance. The claims of VeriSimplePIR performance are described in section 6 and figures 7 & 8 of the paper.

As an example, for a 4 GiB database, VeriSimplePIR has a digest download of 1.56 GiB, a preprocessed ciphertext upload of 17.25 MiB and a preprocessed download of 24.205 MiB. The runtime of the encryption of the client's preprocessed message is roughly 6 seconds, the server's computation of the $\mathbf{Z}$ matrix takes about one minute, and the proof that this computation was performed correctly takes about 16 seconds.

The client then takes about 15 seconds to verify that the result ciphertext is valid, and then another 5 seconds to decrypt the $\mathbf{Z}$ matrix and check this matrix against the original digest.

The client storage in the online phase is 686 MiB. The per query upload and download is about 351 KiB each way. The client's query generation takes about 120 milliseconds, the server takes about 620 milliseconds to generate an answer, the client takes about 6 milliseconds to verify this answer against the preprocessed proof, and then another 33 milliseconds to decrypt the result.

VLHE PIR Performance. The claims of VLHE performance are described in section 6.1.

As an example, for a 4 GiB database, the offline download is about 3.4 GiB. The runtime of the query generation is about 26 milliseconds, and the size of the upload is about 78 KiB. The server takes about 565 milliseconds to generate the encrypted answer and another 10 seconds to generate the proof that this ciphertext was well-formed. The size of the answer ciphertext is about 1.7 MiB, and the size of the proof is about 1.64 MiB (the parameters are chosen so that these values are roughly balanced). The client takes about 3 seconds to verify this proof against the digest and then about 153 milliseconds to decrypt the result.

A.4.2 Experiments

The computational benchmark files both construct a protocol object that is parametrized by a number of database entries N and a database entry bitwidth d. The only other parameters are flags that toggle variation options, including the honest digest assumption. Note that in order to achieve the fastest possible online time for SimplePIR and VeriSimplePIR, the `BASIS` parameter must be set to $\log(p)$ as described in the `README.md`. Note that only changes to `BASIS` that change the number of plaintext elements that can be packed into a single `Elem` type will have an effect on the performance.

VeriSimplePIR Computation Performance [3-5 minutes of compute time, depending on database parameters]: The benchmark of VeriSimplePIR is in the file `bin/demo/bench/preproc_pir_bench`. This benchmark proceeds in three phases. The first is the server computation benchmark. The second is the client computation benchmark. Finally, the online computation of both the client and the server are benchmarked. The split is to reduce the RAM requirement for the offline benchmarks, which do not use the packed database as the operations are not memory-bound.

To take a benchmark of VeriSimplePIR with an honest digest, simply set the `honestHint` flag to `true` when the `VeriSimplePIR` object is constructed.

The bulk of the time for this benchmark is spent on the offline phase, and the file is able to run only the online benchmarks by commenting out the offline benchmark functions in the `main` method.

VLHE PIR Computation Performance [0.5-5 minutes of compute time, depending on database parameters]: The benchmark of VLHE PIR is in the file `bin/demo/bench/pir_bench`. This benchmark simple generates a simulated packed database, then runs through the operations. The `VLHEPIR` class has a flag in the constructor to indicate if the honest digest assumption is active; set this flag to true to take a benchmark with an honest digest.

SimplePIR Computation Performance [0.5-5 minutes of compute time, depending on database parameters]: The benchmarks for our implementation of SimplePIR can be run by toggling the `simplepir` flag in the `VLHE` constructor. We provide a separate benchmark file in `src/demo/bench/simplepir_bench.cpp` for convenience. Note that the textbook SimplePIR protocol uses a 32-bit modulus, which can be set with the `Elem` parameter to `uint32_t` as described in the `README`. The benchmarks for the 64-bit version of SimplePIR can be run by leaving `Elem` as `uint64_t`.

Communication Benchmarks [< 1 minute of compute time]:

We provide a script in `src/demo/scripts/params.cpp` that runs the parameter generation method as a stand-alone function. This function is fully general and can compute parameters for the optimal online communication for VeriSimplePIR, VLHEPIR, and SimplePIR. The option flags are documented in the file. The file can be run by executing `bin/demo/script/params` from the top directory.

Plots: All plots in our paper can be generated in the `plots/` directory. These files were run using `matplotlib` on Python 3.8.

A.5 Notes on Reusability

The compile-time parameters of this artifact are designed to help a user achieve high performance for their specific application. To load a custom database into the library, the `Database` class will need a new constructor to load the custom data. However, we provide a portable multi-precision integer type to store the data values, so this code can natively support data entries that are larger than any machine word.

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/.