USENIX Security '26 Artifact Appendix: PICS: Private Intersection over Committed (and reusable) Sets

Aarushi Goel
Rutgers University

Peihan Miao
Brown University

Phuoc Van Long Pham
Brown University

Satvinder Singh
Purdue University

A Artifact Appendix

A.1 Abstract

This artifact contains the implementation for the protocol in the paper "PICS: Private Intersection over Committed (and reusable) Sets". We re-implement most cryptographic primitives such as OKVS, VOLE and FRI proof here, all with parallelization using the rayon package in Rust.

A.2 Description & Requirements

All tests are ran on 2 machines: one for Sender and one for Receiver. We run our experiments on 2 AWS EC2 instances, with instance type `c5a.16xlarge`. The command to run the experiments can be found on the README file.

A.2.1 Security, privacy, and ethical concerns

Running the program requires quite a lot of RAM. For machines with around 32*GB* of RAM, running the experiments for set size 2^{20} should not be an issue. If there is not enough RAM, the program will get terminated, which may crash the machine. Besides that, there is no other problem.

A.2.2 How to access

We provided a Zenodo URL for our artifact submission: https://zenodo.org/records/17958838. However, as this artifact would likely be an evolving software, we also provide the github link here, and developments should be reflected on the github repository: https://github.com/phuocchubeo123/PICS.git. The code has been tested with Rust 1.84.0.

A.2.3 Hardware dependencies

As the development process was entirely on x86 machine (AWS EC2 c5/c5a instances), it would be recommended to run this library on an x86 machine.

A.2.4 Software dependencies

The software is can be readily ran with Rust installed. However, we use some evolving libraries that might not be backward-compatible, so we provide the list of Github commits for each of these libraries in this section:

- `lambdaworks`: Version `0.14.0`.
- `rand`: Our current code works with version `0.8`. There has been a big refactor in version `0.10` that is not backward-compatible with this code.
- `rand_chacha`: Version `0.9.0`

A.2.5 Benchmarks

In the current version of the artifact, one fresh random seed is shared among two parties at the beginning of the protocol. Then two parties generate two synthetic sets with this seed. The running time of our protocol is quite deterministic given set size, with little variance.

A.3 Set-up

Throughout the development process, we follow the Rust installation guide here: https://www.digitalocean.com/community/tutorials/install-rust-on-ubuntu-linux.

A.3.1 Installation

The evaluator can simply git clone the library and run the binary file with the command in README.

A.3.2 How to run the protocol

By either having two terminals opened, or running on two different machines, the user can run the following command:

```
cargo run --release --bin oprf
(role) (address) (port)
(log_size) (num_threads) (LPN_param)
(sender_commit) (receiver_commit)
```

Each parameter has the following meaning:

- `role`: The role of this machine, either `sender` or `receiver`.

- `address`: For both machines, set this parameter as the IP address of the *receiver*, as we designate the receiver as the TcpListener.
- `port`: The port that is used for communication.
- `log-size`: The logarithm base 2 of two sets' size. For example, `log-size` = 4 means the set size is 16. We recommend setting the set sizes of two parties to be equal.
- `threads`: Number of threads for parallelization.
- `params`: Choosing the LPN parameter for running VOLE. We currently support three sets of LPN parameters: LPN17, LPN21, and LPN25. A parameter set of *LPNXX* means that one can use 2^{XX} VOLE instances, before needing to run `extend` again for another set of fresh 2^{XX} VOLE instances.
- `sender-commit` / `receiver-commit`: Either 0 or 1, to set whether a party runs the committing phase or not.

A.3.3 Basic Test

A run for sets of size 2^{16} would take less than 5 seconds, and should confirm the functionality of the artifact.

A.4 Evaluation workflow

A.4.1 Major Claims

The main major claim of this paper is small commitment size and running time almost linearly dependent on the two sets' size. We tracked the communication cost and print it out when running the binary to support the first claim. For the second claim, we refer to the table in the submitted paper.

A.4.2 Experiments

The following two commands are examples to run the protocol for set sizes 2^{20}, where both sender and receiver commit to their sets. This example experiment runs in the LAN setting, where both parties talk to each other through localhost. Each party parallelizes the computation through 4 cores. First, run on the receiver machine:

```
cargo run --release --bin oprf
receiver 127.0.0.1 8080 20 4 1 1 1
```

Secondly, run on the sender machine:

```
cargo run --release --bin oprf
sender 127.0.0.1 8080 20 4 1 1 1
```

The user should run the receiver's machine first, so that the receiver could set up a Tcp connection, before the sender could connect to it.

Currently, we have computed three sets of LPN parameters, that would work best with the corresponding set sizes provided in Table 1.

Set size	LPN Parameters	Option (`LPN_param`)
2^{16}	`LPN17`	0
2^{20}	`LPN21`	1
2^{21}	`LPN25`	2

Table 1: Suitable LPN parameters for experiments

A.4.3 Setting up the WAN experiment

We use the following script: https://github.com/phuocchubeo123/setup-wan-network.git, which uses the Linux `tc` command to set up the bandwidth (Megabits) and latency (milliseconds) for the experiment.

A.4.4 On choosing LPN parameters

We run a grid search to choose LPN parameters, by running the following repo: https://github.com/phuocchubeo123/LPN-Estimator-Multiprocess.git. The python code that calculates the security level of a set of LPN parameters can be run with the following example command:

```
python home/estimator.py
N=1024 k=652 t=57 q=12 rgular
```

Here, `N`, `k` and `t` are LPN parameters. The field bit size is `q`, for example if `q` $= 12$ then the field is $\mathbb{F}_p$ where $\log p \geq 12$.

The types of noise for LPN that one can choose are `exact` and `rgular`. Here, due to a regex parsing error, one need to use `rgular` instead of `regular` for regular LPN noise. This parsing error appears from the original repo: https://github.com/RabbitCabbage/LPN-Estimator.git. The authors modify this repo to run the security parameter calculation with multiple processes, to speed up the search.

A.5 Notes on Reusability

In the beginning, the authors hope that this artifact would be the first implementation of many cryptographic primitives in Rust. However, we are made aware later that `swanky`[1] has implemented a lot of the cryptographies in Rust. The authors are contemplating on whether to rewrite many portions of this artifact to use swanky directly, and integrate into swanky themselves, to keep this library longer for other researchers in the PSI community to use. If revamping happens, most likely the running time would be faster than the current version of the artifact.

[1] https://github.com/GaloisInc/swanky.git

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/.