TAP: Transparent and Privacy-Preserving Data Services

Daniel Reijsbergen and Aung Maw, *Singapore University of Technology and Design;* Zheng Yang, *Southwest University;* Tien Tuan Anh Dinh and Jianying Zhou, *Singapore University of Technology and Design*

https://www.usenix.org/conference/usenixsecurity23/presentation/reijsbergen

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Artifact Appendices to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

USENIX'23 Artifact Appendix: TAP: Transparent and Privacy-Preserving Data Services

Daniël Reijsbergen[†] Aung Maw[†] Zheng Yang[‡] Tien Tuan Anh Dinh[†] Jianying Zhou[†]

[†]Singapore University of Technology and Design, Singapore, Singapore
[‡]Southwest University, Chongqing, China

A Artifact Appendix

A.1 Abstract

This document describes the code that was used to produce the experimental results in Section 6 of the TAP paper. TAP provides a level of security and privacy that exceeds that of related multi-user approaches (e.g., a trusted server), so the main purpose of the experiments is to demonstrate that TAP still has *practical performance at scale* despite the additional security guarantees. As such, the experiments demonstrate the feasibility (in terms of execution times) of database operations in TAP, e.g., look-up, sum, min, max, and quantile queries.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None – however, if the machine on which the code is run already has MySQL Server installed, then please remember to reset the root password after concluding the experiments.

A.2.2 How to access

The artifact can be found here: https://github.com/tap-group/transparent-data-service/tree/9e97cd42e12fb2941253b0960d4689bf944889a0

A.2.3 Hardware dependencies

The microbenchmark experiments were performed on a MacBook Pro with the following specifications (the code should also work on Linux and Windows systems):

- Processor: 2.4 GHz Quad-Core Intel Core i5
- Memory: 16 GB 2133 MHz LPDDR3
- Operating System: MacOS Monterey Version 12.5.1

The other experiments were performed on Amazon Web Services (AWS) machines: *t2.micro* to represent low-capacity machines, *t2.xlarge* to represent medium-capacity machines, and *m5.4xlarge* to represent high-capacity machines. The latter types have an hourly cost to run, and all types require an AWS account. In the following, we will therefore focus on running the code on a single PC or laptop with the above specifications or similar – we will refer to such a machine as a "medium-capacity machine".

A.2.4 Software dependencies

Working installations of Go, MySQL, and GCC are required (see also the installation guide below). All other dependencies are installed automatically by Go.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Go. The programming code is written in the Go language and therefore requires a working Go installation (version 1.16 or above). To install Go, download it from https://go.dev/doc/install and follow the installation instructions.

MySQL. The TAP implementation requires a working version of MySql Server to simulate the server's back-end, which can be obtained, e.g., through https://dev.mysql.com/downloads/installer/ for Windows. The installer may require that a root password is set. If so, set a temporary password (e.g., '0000'). The TAP code assumes that the root user can access the database without a password. Once MySQL Server has been installed, the password for the root user can be removed as follows: start MySQL from the command line using the following command (which assumes that '0000' is the root user's password)

```
mysql -uroot -p0000
```

then run

```
SET PASSWORD FOR root@localhost='';
```

to remove the password. Finally, create a database named 'tap' by executing the following command in the command line tool:

```
CREATE DATABASE tap;
```

GCC. The TAP implementation uses `go-ethereum` for operations on elliptic curves, and `go-ethereum` in turn requires GCC. To install GCC on Windows, one can use MinGW https://www.mingw-w64.org/downloads/ – make sure that the main executables are accessible via the system path (e.g., by adding `C:\Users\...\mingw64\bin` to the system path variable). On Linux-based systems, run

```
sudo apt install build-essential
```

in the command line.

TAP Code. After downloading the TAP code, use the command line tool to navigate to the main folder and execute

```
go mod tidy
```

to instruct Go to download the required external libraries. It is also helpful to ensure that the *output* folder is empty to avoid confusion with the sample output files that are included with the repository.

A.3.2 Basic Test

To check whether Go was successfully installed, execute the following command on the command line:

```
go version
```

which should return the installed Go version. To check whether MySQL Server was successfully installed, execute the following command:

```
mysql -uroot
```

The above command should start the MySQL command line tool. To check whether GCC was successfully installed, execute the following command:

```
gcc --version
```

This should return the version number of the GCC installation. Finally, to check whether the TAP implementation was successfully built, execute the following command:

```
go run . -experiment1a
```

This starts a basic experiment that tests the time cost of processing data insertions at the server – it should run for less than a minute on a medium-capacity machine. After its completion, it should write "results:" to the command line, followed by a list of simulation results (time and storage costs).

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Look-ups, which are essential for clients who monitor their own data, have negligible overhead, and the cost of the other operations is reasonable on a medium-capacity machine.* In particular, a look-up query in TAP takes ≈0.005s, a sum query takes ≈0.1s, a min/max/quantile query takes ≈1s, and an audit takes ≈10s in a table with 100–10,000 rows. This is demonstrated by Experiment **E1** described in Section 6.2 whose results are displayed in Figure 8a.

(C2): *For look-up and sum queries, total processing times are dominated by proof generation times at the server. For min and quantile queries, proof generation and verification times are similar (around 10 seconds) at the client and server.* This is demonstrated by Experiment **E2** described in Section 6.3 whose results are displayed in Figure 8b.

(C3): *TAP has a smaller storage requirements for the ADS than Merkle^2, although audits and inserts are faster in Merkle^2.* This is demonstrated by experiment **E3** described in Section 6.4 whose results are displayed in Figure 9. (The IntegriDB implementation[1] requires a different set-up – e.g., a specific version of OpenSSL on Linux – and is hence not integrated into the artifact.)

(C4): *On a high-capacity machine, it is feasible to build TAP's data structure, audit (targeted portions of) the data structure, and perform queries even in real-world settings – i.e., databases to which millions of rows are added every hour.* In particular, the time cost of building TAP's data structure increases from ≈0.01s for 100 rows to ≈1000s for 10^7 rows regardless of the number of subtrees. Furthermore, the cost of a full audit increases from ≈1 second for 100 rows to ≈450 seconds for 15 000 rows. Finally, the cost of a quantile query over 1) the entire dataset or 2) a fixed number of subtrees depends at most logarithmically on the total number of rows. This is demonstrated by Experiment **E4** described in Section 6.5 whose results are displayed in Figure 10.

A.4.2 Experiments

(E1): *Microbenchmarks: <1 compute-hour + <1 MB disk*
Execution: This experiment can be reproduced by executing
`go run . -experiment3`
in the repository's main folder. One query type – look-up, sum, average, count, min, max, median, and top 5% queries – is performed in each of experiments 3a–h across 100 epochs.
Results: After each experiment (i.e., 3a–h), a csv-file with a corresponding name will be written to the *output*

[1] https://github.com/integridb/Code

folder – each row in the csv-file corresponds to the result for one epoch. Each csv-file can be used to plot a line in Figure 8a with the epoch counter on the *x*-axis and the query time duration on the *y*-axis. (The line for audits comes independently from a modified version of `experiment7` as discussed under **E4**.) The time costs in the output table should be similar to those mentioned under **C1** on a medium-capacity machine.

(E2): *End-to-end Costs: ≈1 compute-hour + <1 MB disk*

Execution: This experiment can be reproduced by executing

```
go run . -experiment4
```

Results: Each row in the resulting csv-file (*output/experiment4.csv*) corresponds to a unique combination of a) query type (look-up, sum, min, and quantile) and b) number of epochs (10, 30, 100) with 100 new rows per epoch. The following columns contain the relevant data: *prefix_proc_time_server*, *prefix_proc_time_client*, *query_proc_time_server*, *query_proc_time_client*, and *total_time*. The first two correspond to the "prefix tree proof generation" and "prefix tree proof verification" bars in Figure 8b. The "sum tree proof generation", and "sum tree proof verification" bars in Figure 8b represent the difference between the total processing times and the prefix tree processing times. Finally, the "other" bar in Figure 8b represents the difference between the total times and the query processing times. The "other" bar captures network latency, but this is only relevant if the client and server run on different machines.

The claim **C2** can be verified by comparing "query_proc_time_server" to "query_proc_time_client" in the output file: the difference should be more stark in the first six data rows (look-up and sum queries) than in the last six data rows (min and quantile queries)

(E3): *Baselines: 5–10 compute-hours + <1 MB disk*

Execution: This experiment can be reproduced for TAP by executing

```
go run . -experiment5
```

This records the storage cost of building TAP's data structure and the execution cost of the different query types for a varying number of data rows. To reproduce the results for Merkle^2, execute

```
go run .
```

in the `msquare` subfolder of the repository.

Results: After completing the first command, a single output table is created with the data for the graphs of Figure 9 for TAP. The 'storage' column corresponds to Fig 9a, the 'auditor' column to Fig 9b, the 'insert' column to Fig 9c, the 'lookup' column to Fig 9d, the 'sum' column to Fig 9e, and the 'min' column to Fig 9f. The number of rows is not printed by default, but corresponds to 100 times the number in the function call (as per lines 648-654 of `main.go`). The second command will produce a table for Merkle^2 with a similar structure to the one for TAP in the *msquare* folder. **C3** can be verified by comparing the values in the two tables for the same row index.

(Perhaps confusingly, the *x*-axis of Fig. 9a is labeled "epochs" despite there being only one row per epoch.)

(E4): *Scalability: >10 compute-hours + <1 GB disk*

Preparation: The scalability experiments were designed to be run overnight on a *high-capacity machine*, and will take a considerable amount of time to complete with the default settings (i.e., >10 hours on a high-end AWS machine). The range of the experiments can be changed by modifying lines 701 and 704 (the second number in the function call represents the maximum number of epochs, so setting this to a lower number will substantially reduce processing times), and the number of samples can be reduced by setting `nSamples6` and `nSamples7` to 1 in lines 712–713 of `main.go`.

For example, setting `nSamples6` and `nSamples7` to 1, and using

```
getExperimentRange(100, 10000, 3, 10, 100, 2)
```

in line 701 and

```
getExperimentRange(100, 1000, 2, 10, 100, 2)
```

in line 704 should cause both sets of experiments to conclude within 15 minutes on a medium-capacity machine.

Execution: This experiment can be reproduced by executing

```
run . -experiment6
```

for Figures 10a, c, and d, and

```
run . -experiment7
```

for Figure 10b.

Results: The output table consists of one row for each combination of user/sum tree numbers (assuming one sum tree per district), and each cell contains the average result over several queries of the same type. The "quantile_all_total" column contains the results for a quantile query on the entire dataset (Fig. 10c) and "quantile_limited_total" (Fig. 10d) for query over a single subtree. On a medium-capacity machine, **C4** can be verified by observing the trends in the table entries (even if the overall execution times are higher).

A.5 Notes on Reusability

The TAP code can be used to perform queries on data tables with the same format as those generated for the experiments (as specified in *tables/table_factory.go*).

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.