USENIX Security '24 Artifact Appendix: I/O-Efficient Dynamic Searchable Encryption meets Forward & Backward Privacy

Priyanka Mondal
UC Santa Cruz

Javad Ghareh Chamani
HKUST

Ioannis Demertzis
UC Santa Cruz

Dimitrios Papadopoulos
HKUST

A Artifact Appendix

We propose two families of DSE constructions that improve the state-of-the-art, both asymptotically and experimentally, while also addressing the issue of I/O efficiency. In fact, some of our schemes enhance the in-memory performance of previous works. Technically, we revisit and enhance the *lazy de-amortization* DSE construction introduced by Demertzis et al. [NDSS'20], transforming it into an I/O-preserving approach. To demonstrate the feasibility and overhead of our methods, we prototype them and conduct experiments using a standardized set of benchmarks.

A.1 Abstract

This artifact appendix provides source code and build environments for downloading, compiling, and running our schemes presented in the paper. We implemented our prototypes on Linux kernels. The artifact includes SD_a[1C], SD_a[2C], SD_a[NlogN], SD_a[sN], L-SD_d[1C], and L-SD_d[sN], comprising approximately $31K$ lines of C++ code. For evaluation and semantically secure encryption, we utilized OpenSSL-AES PRF. In addition, we employed Oblivious MAP for the SD_d[PiBAS] implementation and merge-sort as the final step in the bucket oblivious sort implementation. Our focus is on the computation time for Search and Update queries, and we measured these parameters for various settings. To simplify the execution process, we provided different input arguments for executing different settings. However, to reproduce the results of our evaluation, the corresponding parameters and configurations must be set before executing the artifact.

A.2 Description & Requirements

Software Requirements. We provide a C++ program that constructs the proposed schemes in the paper. It requires an x86-64 Linux host machine running 64-bit Ubuntu 18.04, g++ v5.5, libssl-dev, make, and nvme-cli packages for compilation and execution. The resulting program accepts various inputs, including the scheme name, disk type, and others. It utilizes a configuration file that specifies the experiment size, such as the database size, number of queries, and result size for each query, for execution. The program establishes a database and measures the execution time for different search/update queries.

Hardware Requirements. The execution of this artifact is resource-intensive and may take a significant amount of time. Our code builds a Linux executable file and utilizes a configuration file to specify various execution settings. The execution requires varying amounts of disk and memory, depending on the target parameters of the experiment. However, the maximum storage needed for the experiments presented in the paper would not exceed 500GB SSD and 2TB HDD disks. Additionally, the required memory would not exceed 128GB for all experiments. For our experiments, we utilized a machine equipped with an Intel Xeon E-2174G 3.8GHz processor.

A.2.1 Security, privacy, and ethical concerns

Our artifact utilizes the "/tmp" folder for data storage and does not access other parts of the file system. It does not utilize the network interface and does not involve any destructive actions.

A.2.2 How to access

We host the artifact on GitHub, which includes all the necessary instructions for compiling and running the program. Additionally, the repository provides the real datasets used in the experiments conducted in the paper.

Repository: https://github.com/jgharehchamani/DSE-with-IO-Locality/tree/fc5942b0d24b7fdc5d8ee4045876d583c812382e

A.2.3 Hardware dependencies

We evaluate our schemes using HDD and SSD hard disks. Additionally, we conduct experiments that utilize memory to showcase the effectiveness of our algorithms. In the experiments involving disk settings, we disabled the cache using two methods: i) disabling the disk cache, and ii) clearing the kernel's cache memory.

These two approaches have been implemented within the code, assuming that the HDD is available at /dev/sda and the SSD is available at /dev/nvme0n1. These configurations can be modified in the Utilities.cpp file.

While we provide and build our schemes, we do not impose any other hardware dependencies apart from a standard x86 host machine required to compile and run the program.

A.2.4 Software dependencies

See A.2 for the list of software dependencies.

A.2.5 Benchmarks

For most of the experiments, we utilized randomly generated datasets with specific parameters mentioned in the paper. These parameters can be set in the configuration file (config.txt), and the code generates the corresponding datasets for execution.

In addition to the random dataset, we also conducted search experiments on a real dataset consisting of 22 attributes and 6,123,276 records of reported crime incidents in Chicago. This dataset is available at the following URL: https://data.cityofchicago.org/public-safety/crimes-2001-to-present/ijzp-q8t2. However, we only used the 5th and 7th attributes, and their corresponding settings and data can be found in the code repository.

A.3 Set-up

A.3.1 Installation

To build the artifact, please refer to the tutorial hosted in the main GitHub repository at README.md.

The steps required for the setup are as follows:

- Download the code repository and build the artifact.
- Configure the target settings in the config.txt file.
- Execute the artifact with the relevant input arguments.

A.3.2 Basic Test

After building the artifact, executing the executable file located in dist/Debug/GNU-Linux with the argument "test" will run a basic test using the amortized PiBAS scheme. The test involves inserting some data and performing a subsequent search.

A.4 Evaluation workflow

We list all our claims and the experiments we performed to support them.

A.4.1 Major Claims

(C1): SD_a[PiBAS] and SD_d[PiBAS] have the worst search performance among all other schemes for large result sizes

(C2): The SD_a[NlogN] and L-SD_d[NlogN] schemes achieve the best search performance in the amortized and de-amortized setting

(C3): SD_a[2C] performs better than SD_a[1C] for small search result sizes

(C4): The search execution time for SD_a[1C] for result sizes bigger than 10^5 remains approximately constant

(C5): When the block size is increased from 32B to 512B, the search time of all schemes increases, but the gap between SD_a[2C]/SD_a[1C] and SD_a[PiBAS] decreases

(C6): Amortized and de-amortized versions of PiBAS, 1C, and NlogN have similar search performance

(C7): The search time of all schemes increases as the database size increases

(C8): All our schemes are significantly faster than SD_a[PiBAS] and SD_d[PiBAS] in the search operation

(C9): When keeping fewer levels than $\log N$, SD_a[NlogN]/L-SD_d[NlogN] outperform PiBAS and 1C based schemes in terms of search time

(C10): As the cache size increases, all schemes' perform better and their search time reduces. However, SD_d[PiBAS] benefits more from the cache than others

(C8): All our schemes outperform SD_d[PiBAS] in search for big enough result sizes (>1K) when the cache is enabled

(C11): When all data is cached (assuming it fits entirely in memory), L-SD_d[NlogN] outperforms other schemes in the search operation

(C12): Our experiments on real datasets show that our schemes clearly outperform both SD_a[PiBAS] and L-SD_d[PiBAS] in search performance, and we reach similar conclusions as synthetic dataset

(C13): The update cost in the amortized schemes depends on the number of previously inserted indexes and it increases when more indexes need to be merged

(C14): SD_a[PiBAS] has the best update cost for small merges and the worst update cost for big merges

(C15): L-SD_d[1C] outperforms other schemes in the update operation for database sizes above $100K$

(C16): L-SD_d[NlogN] has the worst update performance among the de-amortized schemes in the memory setting

(C17): We observe that L-SD_d[1C] has the most efficient update in all database sizes in the HDD setting

A.4.2 Experiments

(E1): *Search Computation Time* [10 human-minutes + 12 compute-hours + 500GB SSD + 2TB HDD]: This set of experiments corresponds to claims **C1**-**C8** and measures the search computation time of all our schemes and the competitors. In these experiments, we vary different parameters, including result size, database size, block size, s parameter, and cache size, and measure the search computation time of each scheme.

How to: Please follow the build and run instructions in Sections A.3.1 and A.3.2.
Execution: The artifact takes three arguments as input (SCHEME_NAME, HARDWARE, CACHE_SIZE). The first parameter determines the scheme type, which can be selected from the following list:
Amortized Schemes: SDa[PiBAS] / SDa[1C] / SDa[2C] / SDa[NlogN] / SDa[3N] / SDa[6N]
DeAmortized Schemes: SDd[PiBAS] / L-SDd[1C] / L-SDd[NlogN] / L-SDd[3N] / L-SDd[6N]
The second parameter indicates the type of memory we want to use, which can be selected among HDD, SSD, and Memory. Finally, CACHE_SIZE denotes the amount of cache memory we want to use to store data in memory. Furthermore, to change the block size of schemes, you need to modify AES_KEY_SIZE in the types.hpp file. Using the above parameters, it is possible to run different types of experiments mentioned in claims **C1**-**C8**.
Results: After running the artifact with the appropriate parameters, it performs some setup steps to build the synthetic dataset. After that, it runs search queries (according to the configuration mentioned in the config.txt file) and measures the search computation time.

(E2): *Real Dataset* [10 human-minutes + 3 compute-hours + 100GB SSD + 100TB HDD]: This experiment corresponds to claim **C12** and measures the search computation time of all our schemes and the competitors on a real dataset (two attributes of the crime dataset).
How to: Please follow the build and run instructions in Sections A.3.1 and A.3.2.
Execution: The execution of the artifact is the same as E1 (A.4.2), except that line 79 needs to be uncommented and line 80 needs to be commented in the main file. This change allows the database generator to use the existing dataset instead of the synthetic one. Additionally, the appropriate configuration in the config.txt file, which is available in the README.md file, needs to be used.
Results: After running the artifact with the appropriate parameters, it performs some setup steps to build the real dataset. After that, it runs search queries (according to the configuration mentioned in the config.txt file) and measures the search computation time.

(E3): *Update Computation Time* [10 human-minutes + 5 compute-hours + 100GB SSD + 100TB HDD]: This experiment corresponds to claims **C13**-**C17** and measures the update computation time of all our schemes and the competitors on a synthetic dataset after setting up the dataset.
How to: Please follow the build and run instructions in Sections A.3.1 and A.3.2.
Execution: The execution of the artifact is the same as E1 (A.4.2). Note that the target dataset and queries should be set in the config.txt file.
Results: After running the artifact with the appropriate parameters, it performs some setup steps to build the synthetic dataset. After that, it runs search queries (according to the configuration mentioned in the config.txt file) and finally executes an update query over the target dataset using the input scheme, measuring the update computation time. Note that the search computation time can be commented out if it is not needed.

A.5 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/.