Menhir: An Oblivious Database with Protection against Access and Volume Pattern Leakage

Leonie Reichert
leonie.reichert@tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Gowri R Chandran
chandran@encrypto.cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Phillipp Schoppmann
schoppmann@google.com
Google
New York, NY, USA

Thomas Schneider
schneider@encrypto.cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Björn Scheuermann
scheuermann@tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

ABSTRACT

Analyzing user data while protecting the privacy of individuals remains a big challenge. Trusted execution environments (TEEs) are a possible solution as they protect processes and Virtual Machines (VMs) against malicious hosts. However, TEEs can leak access patterns to code and to the data being processed. Furthermore, when data is stored in a TEE database, the data volume required to answer a query is another unwanted side channel that contains sensitive information. Both types of information leaks, access patterns and volume patterns, allow for database reconstruction attacks.

In this paper, we present Menhir, an oblivious TEE database that hides access patterns with ORAM guarantees and volume patterns through differential privacy. The database allows range and point queries with SQL-like WHERE-clauses. It builds on the state-of-the-art oblivious AVL tree construction Oblix (S&P'18), which by itself does not protect against volume leakage. We show how volume leakage can be exploited in range queries and improve the construction to mitigate this type of attack. We prove the correctness and obliviousness of Menhir. Our evaluation shows that our approach is feasible and scales well with the number of rows and columns in the database.

CCS CONCEPTS

• **Security and privacy → Database and storage security**; *Privacy-preserving protocols.*

KEYWORDS

Privacy, TEE Database, Oblivious Data Structures, Access Pattern Leakage, Volume Pattern Leakage

ACM Reference Format:

Leonie Reichert, Gowri R Chandran, Phillipp Schoppmann, Thomas Schneider, and Björn Scheuermann. 2024. Menhir: An Oblivious Database with Protection against Access and Volume Pattern Leakage. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3634737.3657005

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Request permissions from owner/author(s).

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0482-6/24/07
https://doi.org/10.1145/3634737.3657005

1 INTRODUCTION

In 2020, Google published statistics on the impact of the COVID-19 pandemic on user mobility to assist the public in mitigating the virus spread. Such studies require large amounts of authentic data [3]. For this purpose, the mobility study relied on data from Google users. By using state-of-the-art differential privacy (DP) mechanisms, the privacy of individuals is protected in published statistics. Despite the resulting limitations on accuracy, the mobility study has found widespread use in epidemiological modeling [13, 54], environmental research [37], policy [4], public health [24], and urban planning [25]. While differential privacy safeguards user privacy in public statistics, the underlying data is often stored centrally. Although this is convenient, it facilitates data breaches [47], which can in turn reduce users' willingness to share data [22, 23].

To ensure privacy, even in the face of data breaches, hardware and software solutions can complement legal measures. Trusted execution environments (TEEs), such as Intel TDX [26], Intel SGX [16], or AMD SEV-SNP [5], are technical approaches that protect processes or virtual machines from malicious hosts in the cloud. However, TEEs rely on the host for paging, which exposes access patterns and control flow [44]. While Oblivious RAM (ORAM) can hide access patterns, running an unpatched ORAM client inside a TEE is insufficient as it leaks control flow information. Oblivious data structures like oblivious AVL trees build on ORAM properties and allow the storage and query of data in a systematic fashion [42, 56]. In this case, query-specific data volume patterns are an additional threat to privacy [31, 33, 36]. Hiding volume patterns is an open challenge in this setting.

In this work, we present Menhir, a privacy-preserving TEE database. It protects against access pattern leakage and volume pattern leakage in a server-only data collection setting. By employing TEE remote attestation, data subjects participating in crowd-sourcing or data donation campaigns can rely on privacy guarantees against compromised data analysts. They can also be sure that no private data is leaked to the server provider.

Menhir requires a server with a TEE such as Intel TDX or AMD SEV-SNP with remote attestation and, depending on the number of columns, at least 2.0 GB of RAM for 2^{20} data points. The approach can also be applied to other TEEs such as Intel SGX. Our evaluation shows that insertion operations to the oblivious database are fast and take around 0.01 ms even on a database with 2^{24} data points.

Menhir allows storing data points with multiple columns in combination with an additional unindexed file with very little impact on query performance. This makes Menhir well suited for crowdsourcing applications such as the aforementioned mobility study [3]. It is also useful for applications where the stored files must be queried based on certain keys. These files can contain sequential data or additional data fields. For example, in a research study on collecting location traces, the indexed columns can contain personal details, while the file itself is a long list of past locations. By adopting Menhir, privacy can be safeguarded while generating insightful location histograms that take into account sensitive information such as infection status or occupation. Furthermore, such files have the potential to accommodate more intricate forms of data, including images or voice samples [11].

Our contributions. The contributions in this paper are summarized as follows:

- We present an oblivious database that supports point and range queries, SQL-like `WHERE`-clauses, and differentially private aggregation. Our construction protects against both access pattern and volume pattern leakage.
- We show how data volume pattern leakage can be used to extract data from the state-of-the-art oblivious AVL tree construction Oblix [42], and how protecting volume patterns with differentially private sanitizers thwarts that attack. Our volume sanitizer improves upon prior work [10] by guaranteeing correctness and requiring fewer dummies for the same DP parameters.
- As part of our construction, we provide a multi-index data structure based on AVL trees that is optimized for ORAMs, while avoiding expensive constructions such as oblivious priority queues. This is of independent interest.
- We prove the correctness and obliviousness of our construction.
- We published our implementation [1] and provide various benchmarks showing its practicality.

Outline. Section 2 introduces the terms and concepts that will be used throughout this paper. Our threat model and an overview of our oblivious database construction are presented in Section 3. In Section 4, we discuss the details of the construction of our oblivious database and improvements on oblivious AVL trees. This is followed by a performance analysis in Section 5. We give an overview of related work in Section 6 and conclude in Section 7. The appendix contains pseudocode as well as the proofs of correctness, obliviousness, and security of our database.

2 PRELIMINARIES

In this section, we provide the background relevant to understanding this paper.

[1]The source code is available at `https://github.com/ReichertL/Menhir`.

2.1 TEEs

Trusted Execution Environments (TEEs) are secure runtime environments to protect code and data from an adversary on the same system. Most relevant for our endeavors are TEEs that isolate virtual machines. To facilitate trusted computing in cloud settings, *remote attestation* is an important feature to establish trust with remote TEEs. It allows verifying that the software running remotely is unaltered.

Intel Trust Domain Extensions (TDX), announced in 2020, is a combination of tools for supporting virtual machine isolation [49]. TDX is designed to guarantee confidentiality and integrity for the memory and CPU state of protected virtual machines called *Trust Domains* (TDs). The TD host cannot access the TD's private memory unless it is explicitly shared by the TD. A TDX module, supplied and signed by Intel, acts as a trusted middleware between the host and TDs [28]. It provides functionalities such as interrupt handling, protects TDs from adversaries by recognizing active attacks, and keeps branch predictions from leaking or being tampered with [27]. The TD owner is responsible for the software in the TD and updates to it and, therefore, needs to be trusted. Similar to its predecessor Intel SGX, a TDX TD relies on the untrusted host for scheduling and paging. To run a TD, the host switches to Secure-Arbitration Mode and calls the TDX module, which can then create, initialize, and schedule TDs. For paging, the host uses an interface of the TDX module for adding and removing TD pages [27].

Unlike SGX, TDX aims to ensure confidentiality and integrity even against side-channel attacks [28]. To mitigate some types of cache side-channel attacks, a single bit is used for each cache line to signify whether it belongs to a TD. In January 2023, TDX was released on the 4th Gen Xeon Scalable CPU platform. As of October 2023, these CPUs are only available through cloud providers to selected customers [6].

AMD SEV-SNP [5] is the third generation of AMD's Secure Encrypted Virtualization (SEV) TEE that provides trusted virtual machines on AMD server CPUs. It was released in 2020 and leverages existing AMD features for trusted computing like hardware memory encryption, the Secure Processor subsystem (AMD SP) for key storage, and encryption of the VM state on world switches. Additionally, SEV-SNP ensures VM memory integrity against a host-level adversary. It also deals with side-channel attacks by restricting interrupts to the trusted VM and protecting branch predictions. Similar to TDX, the host is responsible for scheduling and paging the TEE VM, which provides a side channel for an adversary with the capabilities of the TEE host. Both Intel TDX and AMD SEV do not deal with elaborate hardware adversaries or denial-of-service attacks by the TEE host against the VM. Neither platform prevents leaks via cache-based side channel attacks from code that performs secret-based memory access, e.g., Prime+Probe [5, 29]. This can be solved by careful programming. Providing techniques to this end is the focus of this paper.

2.2 Oblivious Data Structures and ORAMs

An algorithm is *(data-) oblivious* if its control flow and access pattern do not depend on private data. This property is especially useful in cloud computing scenarios. Even if data is encrypted, reconstruction attacks using access patterns can leak sensitive information [33, 36].

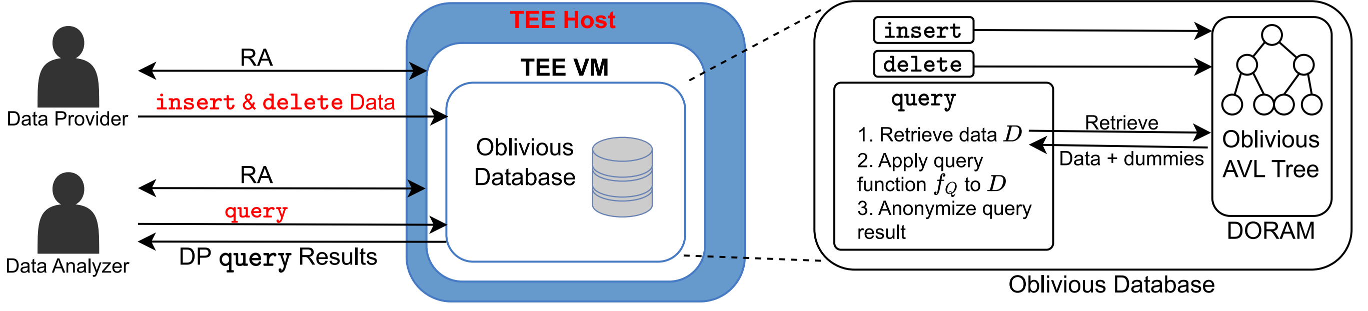


Figure 1: Overview of Menhir. Attack vectors are highlighted in red. RA: Remote Attestation.

Adapting the definition of Wang et al. [56], we define oblivious data structures as follows,

Definition 1. *(Oblivious Data Structure).*
A data structure $\mathcal{D}$ is oblivious if there exists a polynomial time simulator $\mathcal{S}$, such that for any polynomial-length sequence of data structure operations
$\overrightarrow{ops} = ((op_1, args_1), ..., (op_M, args_M))$ *it holds that*

$$addresses_{\mathcal{D}}(\overrightarrow{ops}) \overset{c}{\equiv} \mathcal{S}(\mathcal{L}(\overrightarrow{ops})).$$

where $\overset{c}{\equiv}$ denotes the computational indistinguishability of two distributions. $\mathcal{L}(\overrightarrow{ops})$ is a leakage function. The physical addresses $addresses_{\mathcal{D}}(\overrightarrow{ops})$ are generated by the oblivious data structure during the sequence of operations $\overrightarrow{ops}$.

An *Oblivious RAM* (ORAM) protocol hides the pattern of accesses to private client data stored on an untrusted server [53]. To hide this meta information, data needs to be accessed in an oblivious manner. This means that the access patterns for two sequences of read or write operations must be indistinguishable to the server. ORAM protocols, such as the highly popular PathORAM [53], require a position map and a stash, which are stored on the client side to facilitate lookups.

In distributed settings with untrusted clients, the ORAM client can be placed in a TEE. However, this can only be done if the algorithms of the ORAM client do not leak any access pattern themselves. Such a *Doubly-Oblivious RAM* (DORAM) can be created by altering the protocols for PathORAM to ensure that operations on the stash and the position map are oblivious but still efficient [42]. Oblivious data structures can be built on a DORAM if they have tree-like access patterns [56].

2.3 Differential Privacy

Differential Privacy (DP) is a mechanism for anonymizing responses to database queries by adding noise to the result. Its goal is to prevent an adversary from reconstructing the database or parts of it through strategic queries while allowing aggregate information to be learned about the data. Dwork et al. [17] define (ϵ, δ)-DP as follows:

Definition 2. *(Differential Privacy).*
A randomized algorithm M with domain X is (ϵ, δ)-differentially private whether for all $S \subseteq \mathrm{range}(M)$ and for all data sets $x, y \in X$ differing in at most one item,

$$\Pr[M(x) \in S] \leq \exp(\epsilon) \Pr[M(y) \in S] + \delta.$$

In summary, a mechanism M is DP if an adversary cannot tell whether an arbitrary individual was part of the database or not. This is usually achieved by adding random noise to the outputs of M, which is tailored to the *sensitivity* Δ of M, i.e., the maximal influence any individual input can have on the output.

Important properties of DP include its immunity to post-processing and its composability. Two DP algorithms can be composed with *sequential composition* if they operate on the same data. The new algorithm then provides $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$-DP. In case two DP algorithms operate on disjoint private data sets (i.e., no user is contained in both sets), *parallel composition* can be applied, resulting in $(\max(\epsilon_1, \epsilon_2), \max(\delta_1, \delta_2))$-DP.

Noise Distribution. The most common approach to generate DP noise for continuous values is to draw noise from a Laplace distribution. The scale for the noise is proportional to $\lambda = \Delta/\varepsilon$, i.e., the sensitivity Δ of the query divided by privacy parameter ϵ (also called *privacy budget*). In discrete settings, a *discrete Laplace* distribution is often used, defined by its probability mass function $\Pr[X = x] \propto \exp(-|x|/\lambda)$ [9]. Since both the discrete and continuous Laplace distributions have (theoretically) unbounded support over $(-\infty, \infty)$, they are inconvenient in settings where there are additional requirements on the noise, such as being bound by a certain number or being always positive. By shifting and truncating the distribution appropriately, this can be mitigated at the cost of introducing a non-zero probability δ of providing no privacy to a user (see Definition 2). Following Bell et al. [9], we define the truncated, shifted, discrete Laplace distribution $\mathrm{TSDLap}(t, \lambda)$ with support $\{0, \ldots, 2t\}$ and probability mass function $\propto \exp(-|x - t|/\lambda)$. It can be shown through a tail bound that when choosing
$t = \lceil \Delta + \Delta \ln(2/\delta)/\epsilon \rceil$, adding noise from $\mathrm{TSDLap}(t, \lambda)$ to an output with sensitivity Δ provides (ε, δ)-DP [9].

3 SYSTEM DESIGN OF MENHIR

This section gives a brief overview of Menhir and discusses the security properties of our construction.

3.1 System Overview

An overview of Menhir's architecture and possible attack vectors is given in Figure 1.

Menhir consists of an oblivious database running inside a TEE. As shown in the figure, **data providers** can `insert` or `delete` data

in the database. **Data analyzers** can analyze the collected data by issuing queries. To protect the privacy of data providers, the response to each query q is anonymized with (ϵ, δ)-DP using sequential composition. The database relies on an oblivious AVL tree construction for the underlying data structure (see Section 4.2). This construction supports indexing the collected data over multiple columns, which allows queries to filter by these columns. Additionally, an arbitrary file can be stored for each data point, facilitating a key-value storage with multiple keys. To protect against a malicious TEE host, all database operations hide access patterns and obfuscate the volume of data required to answer a query, the volume pattern.

Each data point, which consists of multiple keys and a value, is stored in one block of a doubly-oblivious ORAM (DORAM, a type of ORAM where accesses to the position map do not leak access patterns). Each block is part of multiple oblivious trees, one tree per database column. This allows the analyzer to write queries that filter the data by different columns. Accesses to the trees are padded to the worst-case tree height to conceal their structure. When processing queries, all relevant data points are retrieved from the ORAM by accessing the root of the oblivious AVL tree corresponding to the database column being queried. Additionally, dummies are retrieved and processed to hide the amount of data required to answer a query. The number of these dummies is determined through (ϵ, δ)-DP. Before returning the (aggregated) result of a query to the data analyzer, it is anonymized with (ϵ, δ)-DP to protect the privacy of data providers.

Menhir extends the previous work Oblix [42] from an oblivious AVL Tree that only protects against access pattern leakage into an oblivious database that supports range and point queries and additionally protects against volume pattern leakage. For this, several essential changes had to be made to the underlying oblivious AVL tree construction. One change is to provide a mechanism for retrieving a fixed number of data points from the oblivious AVL tree, so volume sanitation can be realized. Another change is providing database functionality by constructing multiple AVL trees on the same ORAM nodes. This allows filtering by different columns while minimizing the storage overhead compared to Oblix. Also see Table 1 for a comparison of Menhir's functionalities to Oblix.

For sanitizing the volume patterns, Menhir relies on the findings of Epsolute [10]. However, Menhir improves on the theoretical part of Epsolute by introducing the truncated Laplace function for volume sanitation. This allows dropping the failure probability for volume sanitation which was necessary in Epsolute. Unlike Epsolute, Menhir can ensure that, in all cases, all data points relevant to a query are retrieved and processed.

3.2 Threat Model

3.2.1 Client. The client in Menhir can either be a data provider or a data analyzer. We allow the client to be malicious. This means that if the data provider is malicious, they can perform `insert` or `delete` operations to manipulate the database and the data analysis. By uploading multiple data points, they can skew the evaluation results or mount a denial-of-service attack.

A malicious data analyzer can try to use the `query` interface of Menhir to pose specifically crafted queries in order to reconstruct the database contents. Menhir only allows DP aggregates to be returned by the `query` function to defend against the above attacks. A malicious client, so a data provider or an analyzer, can collude with a malicious TEE Host to infer more information about the data. However, Menhir prevents any leaks sprouting from such collusion by hiding access and volume patterns.

Table 1: Comparison of functionality and privacy guarantees between Oblix [42] and Menhir. (* Only oblivious retrieval of a fixed number of data points).

	Oblix [42]	Menhir (This work)
Oblivious Data Structure	AVL Tree	Database
Query Functionality	Limited*	More comprehensive (Point and Range)
Number of columns that can be indexed	One	Multiple
Access Pattern Leakage mitigated	Yes	Yes
Volume Pattern Leakage mitigated	No	Yes
Volume Sanitation Mechanism	No	Truncated Laplace for (ϵ, δ)-DP
Output Sanitation	No	(ϵ, δ)-DP

3.2.2 TEE Host. In the threat model of Menhir, the main adversary has the capabilities of a TEE Host. The host cannot see the data or code running inside the TEE. However, it can observe the addresses of the accessed data and code at cache line granularity [29]. These access patterns can be used to launch cache-based side channel attacks such as PRIME+PROBE [5, 29]. In Menhir, we provide protection against these attacks using oblivious data structures, such as oblivious AVL trees, that hide access patterns.

Our source code carefully implements the presented algorithms by removing data-dependent branching. To ensure no new branching is reintroduced into the final binary through various compiler optimizations [51], a verified compiler such as CompCert [38] can be used, which supports most languages that follow the ISO C 99 standard. Other steps to solving this issue are turning off most compiler optimizations and programming branch-aware code (e.g., by implementing algorithms that are already data oblivious).

Even if the accesses to private data inside the TEE are carefully obfuscated, the number of accesses to a TEE database can reveal the amount of private data that is processed. This information can be used in database reconstruction attacks mounted by the TEE host [33]. For this, the TEE host needs to know the column and data interval requested by database queries. In Section 4.1.2, an attack is demonstrated which uses this volume pattern leakage. However, for the attack, it is also sufficient if the TEE host only has knowledge about how the private data is distributed to reconstruct parts of the database [36]. We, therefore, assume that the adversary can access the `query` interface of the database. The privacy leakage via maliciously crafted queries that use the `query` interface is mitigated by anonymizing all responses to queries with DP. As the data points which are inserted into the database are uploaded by potentially untrusted data providers, it is also possible for the adversary to access the `insert` and `delete` interfaces of the database.

To defend against database reconstruction attacks using volume patterns, Menhir hides the number of data points processed by a query with DP guarantees.

Although we do not consider timing attacks, the mechanism for volume pattern sanitation makes such attacks more difficult. Denial-of-service and power analysis attacks as well as attacks requiring physical access to the TEE host are out-of-scope of our work.

4 OBLIVIOUS DATABASE: MENHIR

In this section, we present the design of our oblivious database system Menhir. The functionality Menhir provides is a database consisting of a single table with multiple rows and columns that allows for `insert`, `delete`, `query`, and pre-filtering similar to SQL `WHERE`-clauses. While support for multiple tables is feasible, realizing privacy-preserving joins poses its own separate challenges [35, 58]. We leave extending Menhir to multiple tables for future work. The supported `query` types are point queries (where all records with a specific key are retrieved) and range queries (where all records falling into a specific interval are returned). Only differentially private aggregates are returned to the analyzer.

4.1 Querying an Oblivious Tree

Our oblivious database extends the *Doubly-Oblivious Sorted Multimap* (DOSM) construction of Oblix [42], which provides the functionality of a key-value store. We first explain this data structure and discuss in detail the data leakage of this construction. We then present our improved construction that fixes this issue.

4.1.1 Doubly-Oblivious Sorted Multimap (DOSM). Oblivious data structures can be built upon ORAM primitives. Wang et al. [56] show that by replacing pointers in tree-like data structures with pointers to the ORAM, it is possible to make data structures such as AVL trees oblivious. Pointers to child nodes become pointers to the corresponding ORAM block, so $\mathsf{ptr}_i = (ID_i, L_i)$, where ID_i is the ORAM block number and L_i is the corresponding leaf in the ORAM (see Section 2.2). Oblix [42] transfers this AVL tree construction to a doubly-oblivious ORAM to create a DOSM. The DOSM stores key-value pairs by organizing them as nodes in an AVL tree. When computing operations on this tree, the root node, which is stored separately, is used as the entry point. Depending on the operation, the tree is traversed from top to bottom to either find a node with a certain key or determine the correct location to insert a new node. All such operations must be padded to the worst-case tree height h_{max} so that no information is leaked about the structure of the tree. For an AVL tree with n nodes, $h_{max} = 1.44 \cdot \log_2 n$. Balanced trees require rebalancing after insertion and deletion operations. These operations also need to be padded to h_{max} and are not allowed to reveal on which level the insertion or deletion happened. The AVL tree construction allows storing multiple instances of the same key by storing an additional hash to distinguish between data points.

Oblix [42] proposes an algorithm for retrieving several records with the same key. In short, this algorithm, DOSM.FindOblix(k, i, j), retrieves all key-value pairs for key k starting from index i to index j. Although not mentioned in the original paper [42], we can see that it is simple to construct full point queries and range queries from this function: for point queries, instead of using the i-th index, always use index 0, and instead of the j-th index use the highest index possible for key k. Similarly, range queries can be constructed by providing different keys for the start and end of the interval.

4.1.2 Leakage through Volume Patterns. In the following, we show how the construction of the Oblix find function (DOSM.FindOblix) leaks volume pattern, which in turn leaks information about the data points in the database. Following Definition 1 on Page 3, the execution of functions on an oblivious data structure is not allowed to leak any private information to an attacker. Let us assume an attacker $\mathcal{A}$ who can monitor the access patterns to code and data. Query responses to this DOSM are computed using the DOSM.FindOblix function. $\mathcal{A}$ does not need to learn the result of the queries or pose the queries themselves. It is sufficient for them to know what ranges are queried so they can observe the resulting patterns. In particular, $\mathcal{A}$ can see the length of the return array of DOSM.FindOblix and use this information to partially or fully reconstruct the DOSM keys. See Figure 2 for an example. As we see, insertion operations into the return array clearly provide a side channel to the adversary.

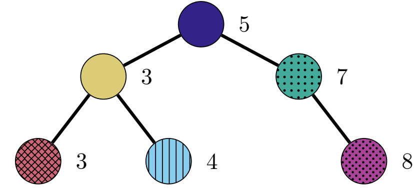


Figure 2: Volume patterns leakage of DOSM.FindOblix. Observing the volume of array R allows the attacker to determine the approximate and exact values of nodes.

4.1.3 DOSM without Volume Pattern Leakage. As we have seen in Section 4.1.2, the DOSM.FindOblix algorithm of Oblix [42] leaks information via volume pattern. To mitigate this leakage, the Find algorithm must not reveal the volume of data used to answer specific queries. This has to include temporary data structures like queues and arrays for which the adversary can observe access patterns and volume patterns.

A naïve and information-theoretically secure solution for hiding the volume of the data such that all queries are completely indistinguishable is to process the entire database every time [43]. However, this is extremely inefficient. Consequently, as a trade-off, some information needs to be revealed to the adversary for better

efficiency. In the context of distributed ORAMs, Bogatov et al. [10] proposed sanitizing the volume patterns through DP algorithms. Here, using all keys, a hierarchical histogram is created which is then perturbed with DP. We employ this idea and determine the number of dummies d used to hide the volume of a query with an (ϵ_s, δ_s)-DP sanitation algorithm S. Improving on the theoretical results of Bogatov et al., Menhir uses a truncated Laplace distribution for sampling noise. This ensures that the number of retrieved data points m is never smaller than the number of nodes n' that need to be retrieved (we see that: $m = n' + d$). The information learned by an attacker who monitors volume patterns is hereby limited through DP. The value of m does not reveal the existence or absence of a node associated with an individual. Optimal parameters for S can be found in [48]. After the data collection has ended and before querying can start, S uses the contents of the DOSM to compute the DP-sanitized volumes for each key. When a query q is posed, the value of m is determined by checking this data structure. For simplicity, we write $m = S(q)$.

To hide the volume pattern, the query function needs to process m values when answering a query. Additionally, the DOSM must be accessed obliviously to not reveal the tree structure. Traversing an AVL tree to sequentially retrieve a fixed number of data points is not straightforward as successors might be located in the right subtree of a node (if it exists) or in a parent node (see Figure 6 in the appendix). Unpadded access to the successor can reveal the tree structure and the level at which the node is located. If all accesses are padded, the overhead of additional accesses to the ORAM becomes large. As a result, only queries with a small selectivity would be possible.

A better approach is to change the structure of the AVL tree so each tree node holds a pointer to its successor. When a new node is inserted, the pointer of its predecessor is replaced with a pointer to the new node. The new node reuses the predecessor's old successor pointer. We now define the functions for retrieving an interval of key-value pairs and for inserting a new key-value pair into the DOSM. The corresponding pseudocode is given in Algorithm 1 and Algorithm 3 in Appendix A.

- DOSM.Find($[k_S, k_E]$,m) $\rightarrow [(k_i, v_i)]_1^m$:
  The algorithm is given an interval from start key k_S to end key k_E as well as a fixed number m of entries to return. This algorithm first traverses the tree to find the smallest node for which $k_S \leq k_i$. The number of accesses is padded to h_{max}. Having found this first node, additional $m - 1$ nodes are retrieved and added to the output by sequentially accessing each node's successor. The algorithm returns a set of key-value pairs of cardinality m.
- DOSM.Insert(k,v) $\rightarrow \perp$:
  The function is given a key k and a value v. First, using k, the tree is traversed starting from the root to find the insertion location of the new node with (k, v). This temporary parent must be a leaf node conforming to the standard AVL insertion strategy. The number of accesses to the ORAM for this step is padded to h_{max}. During traversal, the pointer ptr_{pre} to the predecessor and the pointer ptr_{parent} to the parent are stored. The predecessor is the last node on the way from the root to the leaf, where the path turns to the right child. If the new node is the first node in the tree, no predecessor exists and ptr_{pre} will point to a dummy node (see Figure 6 in the appendix).
  In the next step, the tree is obliviously rebalanced following AVL tree conventions. At last, the successor pointers are updated. If the new node is the first node in the tree, then its ptr_{parent} is used as successor. Otherwise, the new node copies the successor pointer of the predecessor and then sets itself as successor. If a node does not have a successor because it is the last node of the tree, the pointer will point to a dummy node. The dummy node points to itself.

Using the adapted AVL tree, retrieval using DOSM.Find is possible with $O(h_{max} + m)$ ORAM operations and insertion using DOSM.Insert can be done in $O(h_{max})$ ORAM operations.

Correctness. The correctness of the sub-procedure of DOSM.Insert for finding the predecessor of a newly inserted node is given as follows. We call a node smaller than another one if its key is smaller than the key of the other node. In case both keys are equal, the hash associated with each node is used to determine the order. The predecessor of a newly inserted node n_{new} is the largest node which is still smaller than n_{new}. The path from the root to the leaf where n_{new} is inserted consists of a sequence of nodes that are either left or right children. Due to the properties of the binary tree, for all nodes n_r where the path diverges to the right, it holds that $n_r < n_{new}$. This is because all nodes in the right subtree of n_r (where n_{new} is added) are larger than n_r. Any n_r that is found in the right subtree of another n_r is automatically larger and, therefore, better suited as the predecessor for n_{new}. Therefore, the largest node that is still smaller than the new node is the n_r closest to the leaf level. In case the path never diverges to the right and no n_r exists in the path, the new node is the smallest node in the tree and no predecessor exists. Its successor is, therefore, the previous smallest node in the tree. This node is the leaf node that was identified as the insertion location. □

After insertion, a single rebalancing is sufficient to ensure that the AVL Tree invariant is fulfilled. This is due to the fact that all nodes in the tree have balance values of $b \in \{-1, 0, 1\}$ prior to the insertion. The insertion will change this by one. The rebalancing will cause the balance value of the node for which the AVL tree invariant was broken (so $|b| > 1$) to be set to 0. The remaining tree will not become imbalanced if it was balanced before the insertion.

An optimization can be applied during rebalancing. The nodes retrieved during insertion are all nodes from the root to the leaf where the new node is inserted. The rebalancing procedure to be executed depends on the tree's structure. For a left or right rotation, the nodes that need to be updated are the one for which the invariant is broken and one of its children. The imbalance is caused by the newly inserted node. This means both nodes are on the path to the newly inserted node and were retrieved previously. In case a more complex left-right or right-left rotation is required, balancing becomes more difficult. Again the imbalance is caused by the change of subtree heights resulting from the insertion of a new node. Let's take a look at right-left rotations (left-right rotations work analogously). Let n be the node in question, r be the right child of, and rl be the left child of r. Rebalancing procedures that will cause a right-left rotation only occur after insertion to either

Table 2: Information stored in the node of an multi-index AVL tree used to realize an ODB with C columns.

key$_1, \cdots,$ key$_C$		
value		
hash		
Column 1	$\cdots$	Column C
right_child_ptr$_1$	$\cdots$	right_child_ptr$_C$
left_child_ptr$_1$	$\cdots$	left_child_ptr$_C$
left_height$_1$	$\cdots$	left_height$_C$
right_height$_1$	$\cdots$	right_height$_C$
successor$_1$	$\cdots$	successor$_C$

subtree of rl. So both r and rl have been retrieved when the new node was inserted.

4.2 Oblivious Database (ODB)

Our Menhir database can store multiple columns of different types following a column layout schema F. Entries inserted into the database need to follow this schema. It allows for point or range queries along all its indexed columns. Additionally, analyzers can filter along one column. Menhir allows for SQL Queries of the following format

$$\texttt{SELECT}\ f_j(c_f, \epsilon_q)\ \texttt{FROM database WHERE}\ k_S \le c_w \le k_E$$

with query $q \in Q$ is defined as a tuple $q = (k_S, k_E, c_w, c_f, j, \epsilon_q)$. The start key k_S and end key k_E define a range that is used for filtering a column c_w. For the rows that remain after filtering, the values in column c_f are passed to the DP aggregation function f_j. This function is passed in the query via its index j. The function uses privacy budget ϵ_q for anonymization if enough budget is available.

4.2.1 Construction. To create an *Oblivious Database* (ODB), we alter the nodes of the doubly-oblivious sorted multimap (DOSM) (see Section 4.1.3) so that each row of the ODB is represented by one node that is stored in the DORAM. For each of the C columns, a DOSM is built using the same nodes. This requires each node to have C pointers to right children and C pointers to left children. The resulting data structure is a multi-index AVL tree. See Table 2 for an overview of all information stored in an ODB node. Additionally, a total of C root nodes need to be stored as entry points to each DOSM. We define the ODB as follows:

- ODB.Init$(N, F) \rightarrow \perp$:
  This function takes as input a maximum number of entries N and a column schema F with $C = \text{len}(F)$. The function then calculates the required block size for the DORAM using the schema F. Next, it initializes a DORAM using N and the block size. Last, space is allocated for an empty array of length C to store root pointers.
- ODB.Insert$([k_1 \cdots, k_C], v) \rightarrow (\text{ptr}, h)$:
  This function takes as input a set of C keys $[k_1, \cdots, k_C]$ and one value v. The function computes a hash h. It creates a new node using the provided data and the hash. Then the tree structure of each of the C DOSMs is updated iteratively. The function returns a pointer to the newly created node as well as a hash.
- ODB.Find$(k_S, k_E, m, c_w) \rightarrow [[k_{i,1}, \cdots, k_{i,C}], v_i]_1^m$:
  On input of an interval $[k_S, k_E]$, the required number of nodes m, and the index c_w of the column to be queried, this function calls DOSM.Find starting with the root node for column c_w. The function returns the keys and values for m nodes starting from the smallest node for which $k_S \le k_{c_w}$. This function is a sub-procedure of ODB.Query and is not exposed to database analyzers.
- ODB.Query$(k_S, k_E, c_w, c_f, j, \epsilon_q) \rightarrow f_j([k_{i,c_f}]_1^m)$
  On input of a query $q = (k_S, k_E, c_w, c_f, j, \epsilon_q)$, determine the value of m for the interval $[k_S, k_E]$ and column c_w using the corresponding volume pattern sanitizer, so $m = S_{c_w}(q)$. Next, ODB.Find is called with the parameters k_S, k_E, m, and c_w. A set R of rows is returned by the call. Using the data for column c_f of all rows in R[2], function f_j is computed with privacy budget ϵ_q. Then the aggregated and anonymized query result is returned.
- ODB.Delete$(h) \rightarrow \perp$:
  Upon receiving a hash h, the corresponding node n_d is searched. This takes h_{max} accesses. The node n_d is removed from the DORAM. Then, all C DOSMs are updated. For each DOSM c, a replacement is found for k_c of n_d. If n_d does not have any children, no replacement is necessary. If n_d has only one child, this child is the replacement. If n_d has two children, the smallest node of the right subtree is used as a replacement. The search for the replacement is padded to h_{max}, independent of the number of children of the deleted node. Once a replacement is found, all nodes in the DOSM on the path to the deleted node are updated. Again, this operation is padded to h_{max}.

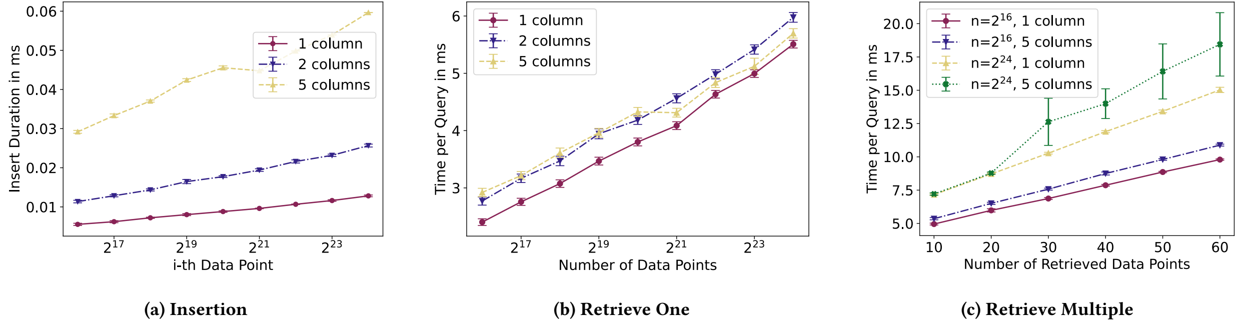
We prove the correctness and obliviousness of the ODB construction in Section B of the appendix.

The GDPR [21] allows people whose data was processed to ask for it to be removed later. A `delete` functionality is therefore required. The hash h is computed from the data uploaded by the data subject or from identifiable information of the data subject. The latter allows the hash to be recreated if the person is no longer in its possession.

4.2.2 Volume Sanitation. To hide volume patterns, m values are retrieved from the database when processing a query. However, not all of these m data points are relevant to the final result for the query (as some are dummies). It is not possible to simply remove all dummies and then pass the array with all relevant values to a DP library, as this would again leak the volume of real data points. Instead, each DP function needs to process all m entries. When a dummy is processed, a dummy operation is made with the neutral element to this operation, e.g., adding a zero for summation. As the data distribution of each column c_i is different, the corresponding volume sanitizer S_i for this column needs to be initiated with suitable parameters. An overview of suitable sanitizers is provided by [48]. In our implementation, we rely on the Epsolute sanitizer [10]. However, by drawing noise from a truncated Laplace distribution, we achieve correctness with a smaller noise overhead compared to Epsolute.

The Epsolute volume sanitizer functions as follows. When initializing the database, the valid data range for each column is passed. For each column, a binary tree with D leaves is created, where D is the size of the (public, discretized) domain of the values in the respective column. Each node at level l in the sanitizer tree, starting with the leaves at $l = 0$, represents a range of 2^l possible values. Each node of this tree is associated with the number of

[2] So k_{c_f} or alternatively v_i, if it contains numeric data.



(a) Insertion

(b) Retrieve One

(c) Retrieve Multiple

Figure 3: Performance of the ODB database for insertion, deletion, and query operations. a) Insertion time of the ith data point for ODBs with different numbers of columns. b) Runtime for retrieving one data point from the ODB. c) Runtime for retrieving a fixed number of data points from the ODB.

database elements falling into the respective range. The volume for any queried range can then be computed by decomposing the query into power-of-2-sized ranges and summing the values associated with the corresponding tree nodes.

To provide differential privacy, the value of each node in the sanitizer tree is perturbed with noise drawn from the Laplace distribution $\mathsf{Lap}(\alpha, \lambda)$, with $\lambda = 1/\varepsilon$ for point queries and $\lambda = \lceil \log_2(D) \rceil / \varepsilon$ for range queries. Here, α is chosen such that drawing D samples guarantees that all samples are positive with probability $1 - \beta$, for negligible β [10, Section 4.6].

We observe that we can guarantee correctness with probability 1 by using a truncated, shifted Laplace distribution $\mathsf{TSDLap}(t, \lambda)$ instead (see Section 2.3). This guarantees (ε, δ)-differential privacy with non-zero probability δ. However, unlike β in Epsolute, the probability δ does not depend on the size of the domain D. Instead, it is only influenced by the $\log_2(D)$ ones drawn for any particular user. In Table 5 in appendix A, we compare the expected number of dummies needed per node for Epsolute and Menhir for a fixed choice of $\varepsilon = \ln(2), \delta = 2^{-20}$.

4.2.3 Output Sanitation. Before the analyzer can pose queries, data points that do not fulfill their quality requirements need to be filtered out. Since data can only be deleted and queries can only be posed afterward, there is no privacy risk for data subjects.

The data collector might be interested in a wide spectrum of information regarding the collected data. A problem is that even if the data collector is honest, they might be compromised or hacked without knowing. Therefore, to protect the privacy of data subjects, only differentially private aggregates are returned. Menhir provides the private aggregation functions `COUNT`, `SUM`, `MEAN`, `VARIANCE`, as well as the `MOST FREQUENT` and `LEAST FREQUENT` item by using the report noisy max algorithm [17]. For a query q, a budget ϵ_q is used for the anonymization. The privacy loss can be quantified using the sequential composition theorem. For a total number n_Q of queries, this means the loss is limited by $\sum_1^{n_Q} \epsilon_q^i$.

The sensitivity for each column is derived from the minimum and maximum values predefined by the attribute schema F. The privacy budget must be maintained by the database system. Data collectors posing queries can, however, specify how much of their budget they want to use for each query. If the budget is used up, no more queries can be processed. While some works reset the privacy budget after a certain time [55], we refrain from this approach as the data in the database does not change after the querying phase starts.

Although the added noise changes the actual result, Bassily et al. [8] have shown that DP can improve the result of statistical analysis. This is because statistics aim to model a real distribution from observed samples and draw knowledge from this real distribution. DP algorithms can improve the generalization error which is introduced by the fact that only a limited number of samples are available.

Non-private databases cover a wide array of functions, such as providing SQL-like `GROUP-BY` functionality, multiple tables, and allowing different `JOIN` functions. Multiple tables can be easily realized with our approach by using a new ODB for each table. However, `JOIN` functions have to ensure that volume patterns remain hidden. As this is a complex task in itself, we point to related work on this topic such as [35, 58]. Similarly, when realizing `GROUP-BY` functions, the number of groups must be either padded to the maximum or sanitized using DP. Wilson et al. [57] discuss how user contribution needs to be limited to provide DP guarantees for SQL-like `GROUP-BY` operations.

5 EVALUATION

In this section, we evaluate the performance and utility of our oblivious database.

5.1 Implementation and Measurements

We implemented the Menhir oblivious database in C++. The implementation uses parts of the Epsolute [10] source code for hiding volume patterns. However, in Epsolute, the queried interval sometimes had to be increased to fit the buckets of the volume sanitizer tree. As a result, additional data points were retrieved due to this padding, which caused significant runtime overheads. Therefore, we adapted the code so that all leaves of the volume sanitizer tree are associated with exactly one element from the data domain instead

Table 3: Comparison of the data protection guarantees and retrieval performance for multiple databases. The values for Oblix were taken from [42]. ORAM speed is the latency for ORAM operations for block size=64 Byte and ORAM size=10^5. Retrieval duration is given for ORAM size=2^{24}.

	Access Pattern Protection	Volume Pattern Protection	ORAM Speed	Retrieval of 10 Data Points		Retrieval of 60 Data Points	
				1 Column	5 Columns	1 Column	5 Columns
Standard	x	x	-	0.107 ms	0.165 ms	0.087 ms	0.082 ms
Naive	✓	✓	90.1 µs	17.4 s	24 s	732.4 s	1065.4 s
Oblix [42]	✓	x	125 µs	≈ 12.5 ms	-	≈ 25 ms	-
Menhir	✓	✓	90.1 µs	7.4 ms	7.4 ms	14 ms	17.5 ms
Speedup (Oblix/Menhir)			1.4×	1.7×		1.8×	

of an interval. The Menhir source code calculates the volume sanitizers S_i once for each column c_i individually. The implementation of the oblivious database can handle integer and float values. When implementing the DP query function, we accounted for leakage when using floating point operations [14].

As the authors of Oblix [42] did not make their DORAM implementation public, we opted for using the readily available Path-ORAM backend of Epsolute. As discussed in the threat model (see Section 3.2), data confidentiality is provided by the TEE. We, therefore, removed the AES encryption for ORAM blocks to improve runtime.

All evaluations were conducted on an AWS server with 121 GB RAM and 16 cores. The selected instance type `r6a.4xlarge` provides 3rd generation AMD EPYC processors (7003-series) for which we enabled the AMD SEV-SNP feature. Each data point represented in the following figures consists of at least 10 measurements. Error bars represent 95 % confidence intervals unless box plots are used. DP noise for volume and output sanitation was drawn from Laplace distributions.

5.2 Performance

Figure 3a shows how fast new elements can be inserted into Menhir's ODB. Using more columns corresponds to an increase in runtime for insertions, with a factor of 2.0 for 2 columns and a factor of 4.64 for 5 columns. This is due to the fact that for each additional column, a separate DOSM needs to be updated. Also, with an increasing amount of data stored in the DOSM and the increasing size of the underlying ORAM, accesses take longer. We can see from the figure that insertion performs well even for a large number of data points and takes less than 0.01 ms for one column even when 2^{24} data points are already in the ODB.

The deletion operation also performs well, despite the large amount of padding that is required to obfuscate the tree structure when deleting a node. For an ODB with 2^{24} data points, deletion of one element takes 45.55 ms for 1 column, respectively 234.15 ms for 5 columns.

For analyzers, it is important to know how fast their queries can be answered. Figure 3b shows how the query answer time is impacted by the number of data points in the ODB. We can see here that the influence of the number of data points stored in the ODB on the runtime is logarithmic, while the impact of the number of columns is constant. The number of points retrieved from the ODB also affects query runtime. Figure 3c shows that the overhead of retrieving increasingly more data points from the ODB is a constant, independent of ORAM size and the number of columns.

5.3 Comparison to Other Databases

To compare against a naive baseline, we implemented a naive database that consists of a list of data points. To query this naive database, first, the number m' of data points to be returned is determined through a sanitized DP histogram. Then an output array with dummy values is initiated with m' slots. Next, for every data point in the database, the naive algorithm iterates over each slot of the output array. If the data point falls into the interval, the first dummy value encountered in the output array is overwritten. The complexity of this algorithm is $O(n \cdot m')$.

In Table 3, we compare Menhir against a standard database without any privacy protection, the naive approach described above, and Oblix [42]. For the standard database without privacy, a MariaDB [41] SQL database was deployed on the AMD SEV-SNP server. Accesses were performed through a Python connector. As we can see from the table, the performance of Menhir is up to 1.8× faster than Oblix while additionally providing volume sanitation guarantees. The speedup in runtime is likely due to differences in implementation and used hardware. Menhir is faster than the naive approach by a factor of 2351 with the same protections and only between 45 times to 219 times slower than the approach without any protection. As we can see, unlike the naive approach, the runtimes of Menhir allow for a real-world deployment.

5.4 Parallelization

The linear increase in Figure 3c can be used to extrapolate the expected time it takes to retrieve a fixed number of data points. For an ODB with one column, 2^{24} data points and relying on an ORAM of the same size, it would take around 10.3 s to retrieve 2^{16} data points, respectively 15.1 s for 5 columns. For an ODB using an ORAM of size 2^{16}, retrieving the same amount of data requires only 6.4 s for 1 column, respectively 7.3 s for 5 columns. This insight can be used to improve the overall performance of Menhir for larger datasets. By storing data in multiple ODBs, which are accessed in parallel (each with a separate ORAM), data points can be retrieved faster and the worst-case runtime for large queries can be capped.

Parallelization in Menhir is realized as follows. Multiple ODBs are associated with the database, each with its' own ORAM of fixed size. New data points are always inserted into the newest DOSM. A

Table 4: Overhead in ms of using multiple OSMs for two different OSM sizes. For each ODB, exactly 60 data points were retrieved in parallel.

Dataset Size	DOSM Size		Factor
	2^{16}	2^{15}	
2^{16}	9.62 ± 0.12	9.2 ± 0.07	1.05
2^{17}	10.19 ± 0.17	10.79 ± 0.13	1.06
2^{18}	11.93 ± 0.16	21.07 ± 0.41	1.77
2^{19}	23.73 ± 0.42	42.03 ± 0.31	1.77
2^{20}	44.73 ± 0.36	74.48 ± 0.93	1.66

new DOSM is created when the maximal capacity of the last one is reached. For deleting a data point, all ODBs have to be checked. To calculate the response for a query, the data returned by all ODBs has to be combined.

Let's take a dataset of size 2^{24}. In the worst case, a query retrieves all data points in the database. Using the extrapolated runtimes from earlier, it can be determined that each ODB should contain a maximum of 2^{16} data points. To hold the complete dataset, 256 ODBs are required. An ODB of size 2^{16} requires 18.36 MB of RAM for 1 column, respectively 58.43 MB for 5 columns.

Parallelization itself also introduces an overhead to runtime due to caching effects. A query is also only as fast as the slowest thread. Table 4 shows the runtime for various dataset sizes and DOSM sizes to help determine the performance overhead introduced by accessing multiple ODBs in parallel. For each ODB, exactly 60 data points were retrieved in parallel. Note that the number of ODBs for different dataset sizes depends on ORAM capacity. We can see that with an increasing total number of data points, the runtime increases despite parallelization. The drastic increase for 2^{18} data points suggests that this is a side effect caused by caching. Therefore, we repeated the measurements on a server with the same number of CPUs but a cache size 32 times as big. The results mirror the measurements from the AMD SEV-SNP server. However, the drastic increase for a DOSM size of 2^{15} is shifted to a dataset size of 2^{20}. It is clear that the improvement of using multiple smaller ODBs is limited by the overhead of parallelization. So despite better worst-case guarantees, the ODB size should be set with considerations for the average case.

To guarantee this performance, each ODB should be associated with one CPU core. It is not uncommon to have up to 64 cores even for consumer CPU. In the case of a dataset with 2^{24} data points, 4 machines with 16 cores each are sufficient to privately query the dataset while guaranteeing one core per OSM. Offloading DOSMs to other machines does induce an overhead as intermediate results have to be communicated over the network. However, if the round delay between the central ODB and the satellite machines is low, parallelization improves the overall runtime.

5.5 Real-World Datasets and Use Cases

In this section, we evaluate the utility of Menhir on real-world datasets and for different use cases. Many datasets used in related work were not relevant for evaluating a database that focuses on volume pattern sanitation, such as the key-value datasets used by Oblix [42]. Others were unavailable, such as the Big Data Benchmark dataset used by Opaque [59]. For this reason, we evaluated Menhir two other datasets[3]. As the TEE server used for measurements only has 16 CPUs, we limited the number of DOSMs to 16, allowing for a maximal dataset size of 2^{20}.

5.5.1 *Real-World Datasets.*

To evaluate the performance of Menhir on realistic data, we used a dataset collected from user requests on the Interplanetary File System (IPFS), a Peer-to-Peer file storage. The data was collected for research purposes and was provided by the authors of [7]. For this evaluation, all sensitive information, such as IP addresses and request IDs, was removed from the data and replaced with pseudonyms. The dataset consists of 1 h of captured IPFS traffic and contains 15.960.697 data points with 8 attributes each. Figure 4a shows the runtime for queries with a selectivity of up to 9% (which are 94.372 data points). Data is stored in ODBs of size 2^{16} and queried in parallel. Additionally to the IPFS dataset, another real-world dataset with more columns was also tested. The Covid-19 dataset [45] contains anonymized information on Mexican Covid-19 patients. It consists of 2^{20} data points and 21 columns.

Figure 4a shows how the query runtime changes for both datasets and different query selectivities. Despite capping the worst-case runtime, parallelization itself does introduce an overhead in the average case.

5.5.2 *Many Columns.*

As seen in Section 5.2, the number of columns does have an impact on the query runtime. Looking at the 20 most voted datasets from `kaggle.com` [30] in the categories "health" and "survey", the median number of columns is 32.5 with a maximum of 644 columns. To analyze the performance of Menhir on a large number of columns, Figure 4b shows the performance of Menhir for datasets with different numbers of columns and 2^{16} data points. The more columns of data are stored, the larger the performance penalties during querying becomes. With less than 1.5 s for a query on a table with 50 columns, our ODB is practical.

5.5.3 *Searchable File Storage.*

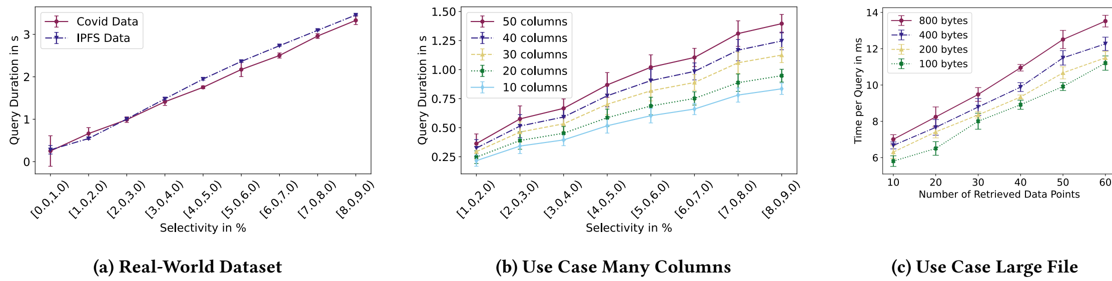
The ODB allows associating each set of keys $[k_1, \cdots, k_C]$ with a value v. The size of this value is set when initializing the ODB. It can be used to associate a file with each tuple of keys or store additional data that does not need to be indexed itself. For all prior evaluations, the size of the value was set to zero. Figure 4c shows how different sizes impact query runtime for a dataset with 2^{16} data points and 1 column. The figure makes clear that the overhead for having values of different sizes associated with each data point is constant. We can see that using Menhir as a searchable file store with volume pattern sanitation is practical.

5.6 Volume Sanitizer Overhead

To hide the response volume of queries, Menhir uses volume pattern sanitizers. To estimate the number of data points that must be retrieved for queries, a sanitizer S_i is computed for each column c_i. The sanitizer S_i is a differentially private histogram for the data domain (see Section 4.2.2). The domain is computed based on the expected maximum and minimum values and the resolution of the data in column c_i.

For point queries, the histogram's data structure is a flat array with as many buckets as there are elements in the domain. To

[3] The datasets can be found at `https://github.com/ReichertL/Menhir`.



(a) Real-World Dataset

(b) Use Case Many Columns

(c) Use Case Large File

Figure 4: Performance of Menhir for different use cases. a) Query duration for different query selectivities on two different real-world datasets (dataset sizes= 2^{20}). b) Query selectivity to query duration for larger numbers of columns (dataset size=2^{16}). c) Time for retrieving a fixed number of datapoints each associated with an unindexed file (dataset size=2^{16}).

deduce the noise required for a specific query, the corresponding bucket is checked to get the sanitized volume m.

For range queries, the data structure of the histogram is a tree. There are two approaches introduced by Epsolute [10] for volume sanitation. The no-γ-method requires that for each OSM, an independent set of sanitizers is created. The γ-method, on the other hand, is optimized for a distributed setting and aims to keep the total noise added as low as possible. Here, even if multiple OSMs are used, only one sanitizer is required for each column.

Figure 5a shows how much noise is added for range queries that cover different ranges but have the same selectivity using the no-*gamma*-method and normal Laplace noise. With the increased range, more buckets in the sanitizer tree are required to cover the queried range. As each bucket adds DP noise to the query, the total noise per query increases.

Figure 5b highlights the relationship between domain size and added noise when using the no-*gamma*-method and normal Laplace noise. Here, all the queries used for this graph have a selectivity of exactly 1% and the range for range queries was fixed to 10. While the noise calculation for both point and range queries depends on the domain size, the influence is clearly visible for range queries but minimal for point queries. However, we can see that the impact of the range of range queries is larger than the impact of the domain size. Both plots, Figure 5a and Figure 5b, clearly emphasize the importance of setting well-suited parameters for the volume pattern sanitizers to best capture the (expected) data distribution of each column. This can be influenced by defining a data resolution. For example, if the expected minimal data resolution is 10, then buckets in the volume pattern sanitizer for any values in between are not necessary. This is especially relevant for floating point data as the volume pattern sanitizer only allows for a limited resolution of fixed size.

5.7 Discussion

As seen in Section 5.2, the runtime of each ODB is linear in the number of elements that fall into the queried interval. Parallelization allows capping the worst-case runtime. This means Menhir is well suited if the expected selectivity of queries is low, for example in heavily distributed or uniformly distributed data. If the database is first filtered by a column containing binary data, the worst-case performance is to be expected. Data resolution and how data is expected to be evaluated are relevant for the decision on how well-suited Menhir is for a specific use case.

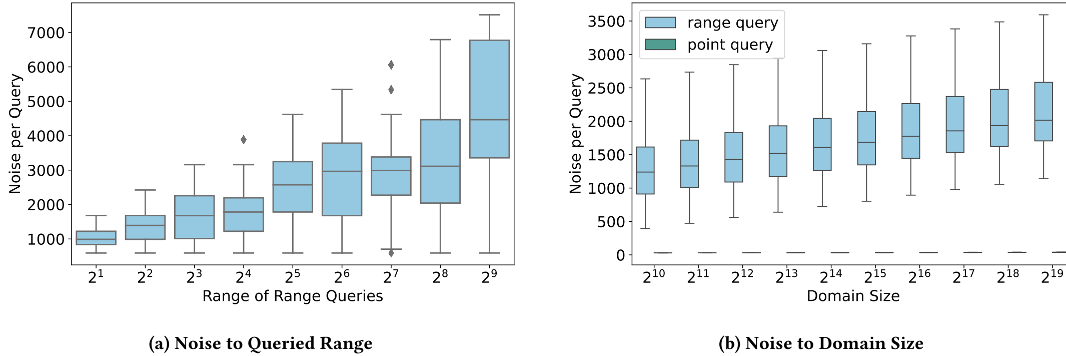
Another potentially interesting use case for Menhir would be an interactive data collection setting where data can be queried while data collection continues. This is possible, yet not optimal, in a setting with a global privacy budget as the privacy budget might already be used up when new, relevant data points are inserted. Queries on these data points will not be answered in a setting with a global budget. Another issue for the interactive setting is the approach Menhir takes on volume pattern sanitation. Every time a query is posed after new data has been added, the volume pattern sanitizers would need to be recalculated. Hence, the privacy budget for sanitation in an interactive setting scales with the number of queries n_Q. We leave the optimization of this bound for an interactive query setting for future work.

6 RELATED WORK

There are many approaches to realizing privacy-preserving data analytics. In this section, we present different approaches to this problem with and without TEEs.

TEEs aim to protect data stored and processed inside as well as the code executed against a malicious host or other types of adversaries on the same system. However, depending on the underlying technology, using a TEE can come with side-channel leakage [44]. At the same time, TEEs provide a great advantage as the performance is better than what is possible with most cryptographic approaches [34]. Also, the guarantee provided by remote attestation can be valuable for systems where trust needs to be well-founded.

A wide range of TEE database systems has been proposed in the past, most of which rely on Intel SGX for data protection. All these approaches provide different database functionalities and security guarantees, especially when it comes to access and volume patterns. ObliDB [18] provides the full range of SQL database functionalities. Encrypted tables are stored outside the SGX enclave in an ORAM and trusted code runs inside the TEE. However, the trusted code does not hide its own access patterns. Also, volume patterns are not considered by this approach. ZeroTrace [50] consists of a secure memory service on top of an ORAM which runs inside an SGX enclave and operates obliviously. However, the memory controller



(a) Noise to Queried Range

(b) Noise to Domain Size

Figure 5: a) Noise applied for range queries depending on the range. The number of data points, the selectivity of queries, and the domain size were fixed. b) Noise applied to range queries and point queries depending on the domain size.

of ZeroTrace does not provide database functionalities. As a result, it does not take volume pattern leakage into account. Oblix [42] uses the methods of Wang et al. [56] to construct an oblivious AVL tree on top of ORAM. The authors pay special attention to the leakage of access patterns from data structures stored inside the TEE. In particular, they discuss how an ORAM client can be changed to not leak access patterns itself. However, they do not consider volume pattern leakage and reveal private information with their query function (see Section 4.1.2). Patel et al. [46] present an approach for volume hiding for encrypted databases without TEE. This approach is similar to the one used in this paper [10] and protects volume patterns with (ϵ, δ)-DP. They do not consider access pattern leakage.

TEEs are not the only approach to preserve privacy during computation and data analysis. *Multi-Party Computation* (MPC) allows two or more parties to evaluate a joint function over the private inputs of the participants [52]. MPC protocols can be realized via generic MPC such as garbled circuits that allow any type of computation, or through special-purpose MPC protocols for specific functionalities. MPC protocols often have quadratic complexity in the number of parties, so they are not suitable for applications with many clients. In these cases, the computation can be outsourced as proposed by Kamara and Raykova [32] and realized by Prio [15]. MPC protocols do not leak access patterns. They generally compute on all available data, as no branching is allowed. They, therefore, do not reveal volume patterns [19].

Encrypted Search Algorithms (ESAs) are another approach to protecting private data in online databases from malicious cloud services [20]. Here, the data is encrypted, so it is unreadable for anyone who does not have the corresponding decryption key while still retaining the capability to search over it without decryption. This can be realized, for example, with ORAMs [53], homomorphic encryption [1], property-preserving encryption, or searchable encryption. However, ESAs are prone to data leakages such as access pattern and volume pattern leakage [31, 33, 36].

A main contribution of this paper is our oblivious multi-index AVL tree with a volume pattern obfuscation. However, other approaches exist for obtaining oblivious algorithms or executables. ObliVM [40] is a domain-specific programming language for writing oblivious algorithms. This approach does not provide protection against volume pattern leakage. GhostRider [39] is a compiler that creates an oblivious executable. For this purpose, it employs ORAM techniques. However, GhostRider requires a CPU with a custom coprocessor making this approach unsuitable for off-the-shelf TEEs. OBFUSCURO [2] is an obfuscation engine that aims to protect intellectual property by using ORAM techniques to protect program code. While access and timing pattern leakage are considered, volume patterns are not covered by this tool.

7 CONCLUSION

In this paper, we presented Menhir, an oblivious database for TEEs such as Intel TDX and AMD SEV-SNP that protects against access pattern leakage and, unlike the previous works, also protects against volume pattern leakage. We first presented an attack against the state-of-the-art oblivious data structure Oblix [42] by using volume pattern leakage. To mitigate this, we changed the underlying AVL tree construction to allow the retrieval of fixed-size intervals from the tree. We build a database on the improved data structure and prove the correctness and obliviousness of our database.

Our evaluation shows that Menhir performs well even for many data points and multiple columns. Larger files can also be associated with each database row, while still retaining good query performance.

8 ACKNOWLEDGMENTS

This project received funding from the ERC under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by the DFG within SFB 1119 CROSSING/236615297 and GRK 2050 Privacy&Trust/251805230.

REFERENCES

[1] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. 2018. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. *Comput. Surveys* 51, 4 (2018), 79:1–79:35.

[2] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, et al. 2019. OBFUSCURO: A Commodity Obfuscation Engine on Intel SGX. In *NDSS '19*. The Internet Society, 1–15.

[3] Ahmet Aktay, Shailesh Bavadekar, Gwen Cossoul, John Davis, Damien Desfontaines, et al. 2020. Google Covid-19 Community Mobility Reports: Anonymization Process Description (version 1.0). *CoRR* abs/2004.04145 (2020). arXiv:2004.04145 https://arxiv.org/abs/2004.04145

[4] Titan Alon, Minki Kim, David Lagakos, and Mitchell VanVuren. 2020. *How should policy responses to the Covid-19 pandemic differ in the developing world?* Technical Report. National Bureau of Economic Research. Accessed: 2023-06-27.

[5] AMD. 2020. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. Accessed: 2023-10-11.

[6] Azure. 2023. Azure Confidential Computing on 4th Gen Intel Xeon Scalable Processors with Intel TDX. https://azure.microsoft.com/en-us/blog/azure-confidential-computing-on-4th-gen-intel-xeon-scalable-processors-with-intel-tdx. Accessed: 2023-06-29.

[7] Leonhard Balduf, Sebastian A. Henningsen, Martin Florian, Sebastian Rust, and Björn Scheuermann. 2022. Monitoring Data Requests in Decentralized Data Storage Systems: A Case Study of IPFS. In *ICDCS'22*. IEEE, 658–668.

[8] Raef Bassily, Kobbi Nissim, Adam Smith, Thomas Steinke, Uri Stemmer, et al. 2016. Algorithmic Stability for Adaptive Data Analysis. In *STOC '16*. ACM, 1046–1059.

[9] James Bell, Adrià Gascón, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, et al. 2022. Distributed, Private, Sparse Histograms in the Two-Server Model. In *CCS 22*. ACM, 307–321.

[10] Dmytro Bogatov, Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2021. ϵpsolute: Efficiently Querying Databases While Providing Differential Privacy. In *CCS '21*. ACM, 2262–2276.

[11] Ferdinand Brasser, Tommaso Frassetto, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, et al. 2018. VoiceGuard: Secure and Private Speech Processing. In *Interspeech '18*. ISCA, 1303–1307.

[12] Jan Camenisch and Anna Lysyanskaya. 2004. Signature Schemes and Anonymous Credentials from Bilinear Maps. In *CRYPTO 04 (Lecture Notes in Computer Science)*, Vol. 3152. Springer, 56–72.

[13] Darlan S Candido, Ingra M Claro, Jaqueline G De Jesus, William M Souza, Filipe RR Moreira, et al. 2020. Evolution and Epidemic Spread of SARS-CoV-2 in Brazil. *Science* 369, 6508 (2020), 1255–1260.

[14] Sílvia Casacuberta, Michael Shoemate, Salil Vadhan, and Connor Wagaman. 2022. Widespread Underestimation of Sensitivity in Differentially Private Libraries and How to Fix It. In *CCS '22*. ACM, 471–484.

[15] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *NSDI '17*. USENIX, 259–282.

[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* (2016). http://eprint.iacr.org/2016/086

[17] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science* 9, 3–4 (2014), 211–407.

[18] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *VLDB Endowment* 13, 2 (2019), 169–183.

[19] Susanne Felsen, Ágnes Kiss, Thomas Schneider, and Christian Weinert. 2019. Secure and Private Function Evaluation with Intel SGX. In *CCSW '19*. ACM, 165–181.

[20] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, et al. 2017. SoK: Cryptographically Protected Database Search. In *S&P '17*. IEEE, 172–191.

[21] GDPR.EU. 2023. What are the GDPR Consent Requirements? https://gdpr.eu/gdpr-consent-requirements. Accessed: 2023-06-29.

[22] Ashish Gupta and Anil Dhami. 2015. Measuring the Impact of Security, Trust and Privacy in Information Sharing: A study on Social Networking Sites. *Journal of Direct, Data and Digital Marketing Practice* 17 (2015), 43–53.

[23] Nick Hajli and Xiaolin Lin. 2016. Exploring the Security of Information Sharing on Social Networking Sites: The role of Perceived Control of Information. *Journal of Business Ethics* 133 (2016), 111–123.

[24] Shima Hamidi and Ahoura Zandiatashbar. 2021. Compact Development and Adherence to Stay-at-Home Order During the Covid-19 Pandemic: A Longitudinal Investigation in the United States. *Landscape and Urban Planning* 205 (2021), 103952.

[25] Marc Hasselwander, Tiago Tamagusko, Joao F Bigotte, Adelino Ferreira, Alvin Mejia, et al. 2021. Building Back Better: The Covid-19 Pandemic and Transport Policy Implications for a Developing Megacity. *Sustainable Cities and Society* 69 (2021), 102864.

[26] Intel. 2020. Intel® TDX. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html. Accessed: 2023-06-14.

[27] Intel. 2021. *Architecture Specification: Intel® Trust Domain Extensions (Intel® TDX) Module.* Technical Report. Intel. Accessed: 2023-06-14.

[28] Intel. 2021. *Intel® TDX Module Base Architecture Specification.* Technical Report. Intel. Accessed: 2023-06-14.

[29] Intel. 2023. MKTME Side Channel Impact on Intel TDX. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/mktme-side-channel-impact-on-intel-tdx.html. Accessed: 2023-08-08.

[30] Kaggle Inc. 2023. kaggle. https://www.kaggle.com. Accessed: 2023-06-13.

[31] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Thomas Schneider, Amos Treiber, et al. 2022. SoK: Cryptanalysis of Encrypted Search with LEAKER - A framework for LEakage AttacK Evaluation on Real-world data. In *EuroS&P '22*. IEEE, 90–108.

[32] Seny Kamara and Mariana Raykova. 2011. Secure Outsourced Computation in a Multi-Tenant Cloud. In *IBM Workshop on Cryptography and Security in Clouds*. IBM, 1–5.

[33] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2016. Generic Attacks on Secure Outsourced Databases. In *CCS '16*. ACM, 1329–1340.

[34] Patrick Koeberl, Vinay Phegade, Anand Rajan, Thomas Schneider, Steffen Schulz, et al. 2015. Time to Rethink: Trust Brokerage Using Trusted Execution Environments. In *TRUST '15*. Springer, 181–190.

[35] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *VLDB Endowment* 13, 11 (2020), 2132–2145.

[36] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage. In *S&P '18*. IEEE, 297–314.

[37] Thomas Lecocq, Stephen P Hicks, Koen Van Noten, Kasper Van Wijk, Paula Koelemeijer, et al. 2020. Global Quieting of High-Frequency Seismic Noise due to Covid-19 Pandemic Lockdown Measures. *Science* 369, 6509 (2020), 1338–1343.

[38] Xavier Leroy. 2023. CompCert. https://compcert.org/index.html. Accessed: 2023-10-11.

[39] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, et al. 2015. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *ASPLOS '15*. ACM, 87–101.

[40] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *S&P '15*. IEEE, 359–376.

[41] MariaDB Foundation. 2023. MariaDB Server: The Open Source Relational Database. https://mariadb.org/. Accessed: 2023-12-05.

[42] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. 2018. Oblix: An Efficient Oblivious Search Index. In *S&P '18*. IEEE, 279–296.

[43] Muhammad Naveed. 2015. The Fallacy of Composition of Oblivious RAM and Searchable Encryption. *IACR Cryptol. ePrint Arch.* (2015). http://eprint.iacr.org/2015/668

[44] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. 2020. A Survey of Published Attacks on Intel SGX. *CoRR* abs/2006.13598 (2020). arXiv:2006.13598 https://arxiv.org/abs/2006.13598

[45] Meir Nizri. 2023. COVID-19 Dataset. https://www.kaggle.com/datasets/meirnizri/covid19-dataset. Accessed: 2023-10-16.

[46] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing. In *CCS '19*. ACM, 79–93.

[47] Privacy Rights Clearinghouse. 2023. Data Breach Chronology. https://privacyrights.org/data-breaches. Accessed: 2023-06-21.

[48] Wahbeh H. Qardaji, Weining Yang, and Ninghui Li. 2013. Understanding Hierarchical Methods for Differentially Private Histograms. *VLDB Endowment* 6, 14 (2013), 1954–1965.

[49] Muhammad Usama Sardar, Saidgani Musaev, and Christof Fetzer. 2021. Demystifying Attestation in Intel Trust Domain Extensions via Formal Verification. *IEEE Access* 9 (2021), 83067–83079.

[50] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. 2018. ZeroTrace: Oblivious Memory Primitives from Intel SGX. In *NDSS '18*. The Internet Society.

[51] Laurent Simon, David Chisnall, and Ross J. Anderson. 2018. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *EuroS&P '18*. IEEE, 1–15.

[52] Nigel P. Smart. 2016. *Cryptography Made Simple*. Springer. https://doi.org/10.1007/978-3-319-21936-3

[53] Emil Stefanov, Marten Van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, et al. 2018. Path ORAM: An Extremely Simple Oblivious RAM Protocol. *Journal of the ACM* 65, 4 (2018), 1–26.

[54] Mihaly Sulyok and Mark Walker. 2020. Community Movement and Covid-19: A Global Study Using Google's Community Mobility Reports. *Epidemiology & Infection* 148 (2020).

[55] Jun Tang, Aleksandra Korolova, Xiaolong Bai, Xueqiang Wang, and XiaoFeng Wang. 2017. Privacy Loss in Apple's Implementation of Differential Privacy on MacOS 10.12. *CoRR* abs/1709.02753 (2017). arXiv:1709.02753 http://arxiv.org/abs/1709.02753

[56] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, et al. 2014. Oblivious Data Structures. In *CCS '14*. ACM, 215–226.

[57] Royce J. Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, et al. 2020. Differentially Private SQL with Bounded User Contribution. *PoPETS* 2020, 2 (2020), 230–250.

[58] Xingliang Yuan, Xinyu Wang, Cong Wang, Chenyun Yu, and Sarana Nutanong. 2017. Privacy-Preserving Similarity Joins Over Encrypted Data. *IEEE Transactions on Information Forensics and Security* 12, 11 (2017), 2763–2775.

[59] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, et al. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI 2017*. USENIX, 283–298.

A ALGORITHM

In this appendix, we provide and explain the pseudocodes referenced in Section 4.1.3 and Section 4.2. We use lC and rC as shorthand for "left Child" and "right Child", the successor of node i is written as $node_i.succ$, and bT stands for "balanceType".

Algorithm 1 DOSM.Insert(k, v)

```
h ← hash(k, v)
node_new, ptr_new ← AVLTreeNode(k, v, h)
nodes ← [ ]
ptr_pre, ptr_parent ← ptr_dummy
ptr_i ← ptr_root
//Find insert location and predecessor
for i ← 1 to h_max do
    node_i ← ORAM.Get(ptr_i)
    nodes.append(node_i)
    left ← (k < k_i) ∨ (k == k_i ∧ h < h_i)
    ptr_pre ← if not left then ptr_i
    ptr_{i+1} ← if left then node_i.lC else node_i.rC
    isDummy ← (ptr_{i+1} == ptr_dummy)
    ptr_parent ← if not isDummy then ptr_i
    ptr_i ← ptr_{i+1}
end for
//update parents (h_max write operations to ORAM)
bT, ptr_child, ptr_grandchild ← UpdateParents(nodes)
//Update successor pointers
node_new ← ORAM.Get(ptr_new)
node_pre ← ORAM.Get(ptr_pre)
isSmallest ← (ptr_pre == ptr_dummy)
node_pre.succ ← if not isSmallest then ptr_new
node_new.succ ← if isSmallest then ptr_parent else ptr_pre
ORAM.Put({node_pre, node_new})
//Insert node and rebalance tree
Rebalance(bT, ptr_new, ptr_child, ptr_grandchild)
return {h_i, ptr_new}
```

Algorithm 1 describes how a key-value pair $[k, v]$ can be inserted into a doubly-oblivious sorted multimap (DOSM) without leaking the structure of the tree. Conditions like "if c then r=a else r=b" can be realized without jumps as a single mathematical statement of the form: $r = c \cdot a + (\text{not } c) \cdot b$. The simpler version of this condition "if c then r=a" can be realized as: $r = c \cdot a + (\text{not } c) \cdot r$.

The sub-procedure Rebalance() relies on the insight in Section 4.1.3 that only a single or double rotation is sufficient to ensure that the AVL tree invariant is fulfilled after a new node is inserted. It performs 3 read and 3 write operations to ORAM independent of the fact whether any rebalancing is needed. Algorithm 2 shows the algorithm for the sub-procedure.

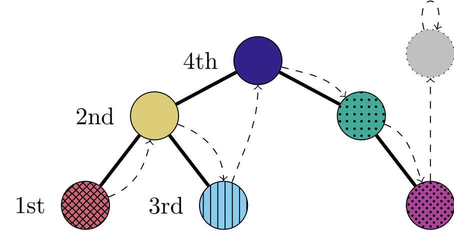


Figure 6: Order in which nodes are accessed for the interval $[2-4]$ and $m = 4$. Pointers to successor nodes are shown with dashed lines.

Algorithm 2 Rebalance($bT, ptr_{new}, ptr_{child}, ptr_{grandchild}$)

```
node_new ← ORAM.Get(ptr_new)
node_c ← ORAM.Get(ptr_child)
node_g ← ORAM.Get(ptr_grandchild)
//depending on balanceType bT these rotations are only dummy operations
Rotate(node_new, node_c, bT)
Rotate(node_new, node_g, bT)
ORAM.Put({node_new, node_c, node_g})
return {h_i, ptr_new}
```

The insertion procedure sets the successor of each newly inserted node. This is relevant for data retrieval. Figure 6 shows an example in which order elements in an AVL Tree are accessed using the pointer to the next node when an interval is retrieved. Algorithm 3 explains in detail how data is retrieved from the DOSM for an interval $[k_S, k_E]$ and a fixed number m. This m is selected to hide the volume of data in the queried interval. This algorithm returns a set of m nodes, each with all its associated data (keys and values). The function FindSmallesNodeInInterval() retrieves the smallest node for which the key k is larger or equal to k_S. If multiple nodes with the same key exist, they are ordered based on their hash. The function always makes h_{max} accesses to the ORAM.

Algorithm 3 DOSM.Find(k_S, k_E, m)

```
ptr_root ← DOSM.root
node_i ← FindSmallestNodeInInterval(ptr_root, k_S, k_E)
for i ← 1 to m − 1 do
    R ← R ⋃ {node_i}
    node_{i+1} ← node_i.succ
end for
return R
```

In Section 4.2, an oblivious database (ODB) is built from the DOSM. Algorithm 4 explains how a data collector can query the ODB. During data retrieval and computation, no volume patterns are leaked. The function takes as input a query consisting of an interval $[k_S, k_E]$, the column index c_w for which entries are retrieved (the column for the WHERE-clause), and the column index c_f for which the function f_j is applied on the retrieved entries. Function f_j is differentially private and is passed by the query through

$\log_2(D)$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Epsolute [10]	21	44	69	96	125	156	189	224	261	300	341	384	429	476	525	576	630	685	742	800
Menhir	22	45	69	93	118	143	168	193	219	245	271	297	323	349	375	401	428	455	481	508
ratio	1.05	1.02	1.00	0.97	0.94	0.92	0.89	0.86	0.84	0.82	0.79	0.77	0.75	0.73	0.71	0.70	0.68	0.66	0.65	0.64

Table 5: The size of the sanitized domain D to the expected number of dummies needed for volume sanitation by Epsolute [10] and Menhir, as well as their ratio (lower is better). $\varepsilon = \ln 2, \delta = 2^{-20}$.

its index j. It uses the privacy budget ϵ_q for computing the differentially private aggregate. If the remaining global privacy budget is less than ϵ_q, the query can not be answered.

Algorithm 4 ODB.Query($k_S, k_E, c_w, c_f, j, \epsilon_q$)

```
m ← S_{c_w}(k_S, k_E)
R ← ODB.Find(k_S, k_E, m, c_w)
value ← f_j(c_f, ε_q, R)
return value
```

ODB.Query uses the find function from Algorithm 5 for retrieving m data points for the respective interval $[k_S, k_E]$ from the column c_w. For this purpose, it first finds the smallest node larger than or equal to k_S from the DOSM for column c_w. For each column, the ODB holds a pointer to the root node of the corresponding DOSM. Due to the use of the truncated Laplace function, all data points that fall into the interval $[k_S, k_E]$ are contained in the output of ODB.Find.

Algorithm 5 ODB.Find(k_S, k_E, m, c_w)

```
ptr_root ← ODB.DOSMRoots[c_w]
node_i ← FindSmallestNodeInInterval(ptr_root, k_S, k_E)
for i ← 1 to m − 1 do
    R ← R ⋃ {node_i}
    node_{i+1} ← node_i.succ
end for
return R
```

Using a truncated Laplace distribution for volume sanitation not only allows us to drop the failure probability required in Epsolute [10]. The amount of noise that needs to be introduced is also reduced. In Table 5, we compare the expected number of dummies needed for volume sanitation per node for Epsolute [10] and Menhir for a fixed choice of $\varepsilon = \ln 2, \delta = 2^{-20}$.

B CORRECTNESS AND OBLIVIOUSNESS

In the following, we prove the correctness and obliviousness of our ODB construction.

B.1 Correctness

Definition 3. *(Correctness).*
Let $x \in \{0,1\}^*$ represent the contents of a database table with multiple rows and columns. Function f is an operation on this table. A protocol π implementing f is correct *if the output of π is computationally indistinguishable from $f(x)$. In short, $output^{\pi}(x) \overset{c}{\equiv} f(x)$. This means for a negligible function μ it holds that*

$$Pr[output^{\pi}(x) = f(x)] \geq 1 - \mu.$$

We first introduce some notations. As mentioned before, a query $q \in Q$ is a tuple $q = (k_S, k_E, c_w, c_f, j, \epsilon_q)$. Let V_q be the set containing all entries from column c_f for which the corresponding entry in column c_w fulfills the query condition $k_S \leq c_w \leq k_E$. For proper privacy protection, we require that the sensitivity of f_j is set correctly for the data type in column c_f.

Theorem 1. *Following Definition 3, the ODB scheme in Section 4 is correct for the functions* ODB.Init, ODB.Insert *and* ODB.Delete.

Proof. This follows from the correctness of the oblivious data structure framework of Wang et al. [56] and the plaintext AVL tree construction which the DOSM builds on. □

The correctness of the ODB.Query function boils down to the correctness of the ODB.Find function and the correctness of the function f_j evaluation. First, we discuss the correctness of the ODB.Find function.

Theorem 2. ODB.Find *correctly returns all elements in an interval given by a query $q \in Q$ if $m \geq |V_q|$.*

Proof. The ODB.Find function makes a call to the DOSM.Find function for finding the elements in a given interval. Therefore, the correctness of the ODB.Find function follows from the correctness of the DOSM.Find function for a particular column c_f. □

Theorem 3. *The* ODB.Query *function correctly computes the function f on the required entries.*

Proof. As mentioned earlier, the correctness of the ODB.Query function depends on the correctness of the ODB.Find function and the computation of function f. Theorem 2 proves the correctness of ODB.Find. To show that the query function f is computed correctly, it suffices to show that (a) all required elements are included in the aggregation and (b) the dummies added through the volume sanitizer do not change the output. Claim (a) follows directly from the fact that our volume sanitizer always adds positive noise through the truncated Laplace mechanism (see Section 4.2.2). For this reason, the constraint $m \geq |V_q|$ always holds. This is in contrast to Epsolute [10], which fails with a (negligible) probability β. Claim (b) can be ensured to hold by using the neutral element of the respective aggregation function as the value for dummies, thereby ensuring that they don't change the output.

□

Remark: A malicious database provider can insert data points in the ODB and delete these. This allows them to alter the result of

all queries and skew the data analysis. However, this is a general risk of crowd-sourcing campaigns, and in particular, it does not depend on or reveal any user's input. Sybil attacks can be made more resource intensive for attackers by requiring data providers to identify themselves. To preserve privacy, the identification process can be realized through anonymous authentication schemes. However, such schemes are not the focus of this work, so we point the reader to the relevant work on this topic, such as [12].

B.2 Obliviousness

THEOREM 4. *The* ODB.Init*,* ODB.Insert*, and* ODB.Delete *operations of the ODB scheme are oblivious with a leakage function* $\mathcal{L} = ((op_1, c_1) \cdots, (op_M, c_M))$ *according to Definition 1 on page 3. The leakage function leaks only the operation type* op_i*, the accessed column* c_i*, and the total number of operations* M*, but nothing else.*

PROOF. First, observe that ODB.Init, ODB.Insert, and ODB.Delete make the same number and types of ORAM accesses for two function calls even if different data is provided. It then follows immediately from the security of the underlying ORAM scheme [53] that the memory addresses produced are indistinguishable for any two function calls. We can, therefore, define the simulator $\mathcal{S}$ that takes the sequence of operations and then runs the corresponding algorithms on a dummy index-value pair (say, $(0, 0)$).

□

Without additional perturbation, the volume of data used for answering a query can be used to reconstruct a database [33, 36]. Algorithms for oblivious databases that do not pay attention to this side-channel, such as Oblix [42], end up leaking this information (see Section 4.1.2). To obliviously answer queries, one could process the whole database for each query. However, this is not very efficient. Therefore, we weaken the definition by allowing additional leakage per operation but requiring that this leakage be differentially private.

DEFINITION 4. *(Obliviousness with DP volume leakage). Let* m_i *be the volume of data processed for an operation* op_i *on column* c_i *with arguments* arg_i*. A data structure* $\mathcal{D}$ *is oblivious with DP volume leakage, if there exists a polynomial time simulator* $\mathcal{S}$*, such that for any polynomial-length sequence of data structure operations* $\overrightarrow{ops} = ((op_1, c_1, args_1, m_1), ..., (op_M, c_M, args_M, m_M))$ *and* $\mathcal{L}(\overrightarrow{ops}) = ((op_1, c_1, m_1), \cdots, (op_M, c_M, m_M))$ *it holds that*

$$addresses_{\mathcal{D}}(\overrightarrow{ops}) \overset{c}{\equiv} \mathcal{S}(\mathcal{L}(\overrightarrow{ops})),$$

and each m_i *provides* (ε, δ)*-differential privacy with respect to individual database items.*

THEOREM 5. *For an ODB with* (ε, δ)*-DP volume sanitation, a sequence of* M ODB.Query *operations is oblivious with* $(M \cdot \epsilon, M \cdot \delta)$*-DP volume leakage.*

PROOF. We start by defining the simulator $\mathcal{S}$ that takes as input the leakage $\mathcal{L}$ containing the operations op_i, sanitized volumes m_i, and column indices c_i, then calls ODB.Query on c_i and a dummy key, replacing m in the first line by m_i.

Since FindSmallestNodeInInterval() makes the same number of ORAM queries independent of the arguments, the resulting addresses will be indistinguishable from the real implementation of ODB.Query.

It remains to be shown that each m_i is differentially private. As described in Section 4.2.2, the volume sanitizer uses a binary tree, where each database item is counted exactly once per level. If we consider two neighboring databases that differ in exactly one item x, there will therefore be exactly $h = \lceil \log_2(D) \rceil$ nodes that x contributes to. Since each node has noise drawn from $\mathsf{TSDLap}(t, h/\varepsilon)$, revealing a single node's value is $(\varepsilon/h, \delta/h)$-DP for appropriately chosen t (see Section 2.3). By basic composition, the revealing all h nodes is, therefore, (ε, δ)-DP. The claim follows through basic composition across M queries.

□