SWAT: A System-Wide Approach to Tunable Leakage Mitigation in Encrypted Data Stores

Leqian Zheng
City University of Hong Kong
leqian.zheng@my.cityu.edu.hk

Lei Xu
Nanjing University of Science and Technology
xuleicrypto@gmail.com

Cong Wang
City University of Hong Kong
congwang@cityu.edu.hk

Sheng Wang
Alibaba Group
sh.wang@alibaba-inc.com

Yuke Hu
Zhejiang University
The State Key Laboratory of Blockchain and Data Security
yukehu@zju.edu.cn

Zhan Qin
Zhejiang University
The State Key Laboratory of Blockchain and Data Security
qinzhan@zju.edu.cn

Feifei Li
Alibaba Group
lifeifei@alibaba-inc.com

Kui Ren
Zhejiang University
The State Key Laboratory of Blockchain and Data Security
kuiren@zju.edu.cn

ABSTRACT

Numerous studies have underscored the significant privacy risks associated with various leakage patterns in encrypted data stores. While many solutions have been proposed to mitigate these leakages, they either (1) incur substantial overheads, (2) focus on specific subsets of leakage patterns, or (3) apply the same security notion across various workloads, thereby impeding the attainment of fine-tuned privacy-efficiency trade-offs. In light of various detrimental leakage patterns, this paper starts with an investigation into which specific leakage patterns require our focus in the contexts of key-value, range-query, and dynamic workloads, respectively. Subsequently, we introduce new security notions tailored to the specific privacy requirements of these workloads. Accordingly, we propose and instantiate Swat, an efficient construction that progressively enables these workloads, while provably mitigating system-wide leakage via a suite of algorithms with tunable privacy-efficiency trade-offs. We conducted extensive experiments and compiled a detailed result analysis, showing the efficiency of our solution. Swat is about an order of magnitude slower than an encryption-only data store that reveals various leakage patterns and is two orders of magnitude faster than a trivial zero-leakage solution. Meanwhile, the performance of Swat remains highly competitive compared to other designs that mitigate specific types of leakage.

PVLDB Reference Format:
Leqian Zheng, Lei Xu, Cong Wang, Sheng Wang, Yuke Hu, Zhan Qin, Feifei Li, and Kui Ren. SWAT: A System-Wide Approach to Tunable Leakage Mitigation in Encrypted Data Stores. PVLDB, 17(10): 2445 - 2458, 2024.
doi:10.14778/3675034.3675038

*Cong Wang is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 17, No. 10 ISSN 2150-8097.

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CongGroup/SWAT.

1 INTRODUCTION

With the advent of cloud computing, many companies and institutions are outsourcing databases and workloads from their private data centers to the cloud. While this transition offers advantages such as increased availability, scalability, and cost-effectiveness, it also exposes users to potential privacy breaches and data abuse. Consequently, there has been significant progress in constructing *encrypted databases* to prevent adversaries with high privileges or even physical access to the server from learning sensitive data. Specifically, one line of work [57, 58] leverages specialized cryptographic primitives to perform different operations over ciphertexts. Another prosperous line of work [2, 4, 21, 59, 73, 76, 83, 90] we will follow utilizes trusted execution environments (TEE), *e.g.*, Intel SGX and AMD SEV, to process confidential data as plaintext in an isolated and protected approach.

Unfortunately, despite powerful enclaves, recent studies show that encrypted databases still exhibit diverse leakage patterns, including 1) memory access pattern indicates which memory blocks are accessed, potentially revealing sensitive information like the access frequency of encrypted records; 2) volume pattern refers to the size of the query result set, which can be easily obtained by observing network communication; 3) order pattern refers to the ordinal relationship between data, which may be inferred from data storage (*e.g.*, encrypted albeit sequentially stored data) or memory accesses (*e.g.*, a search over a B+ tree directly reveals the exact ordinal relationship between nodes along the path); 4) query correlation pattern reveals how queries are correlated, *e.g.*, humans or workloads often generate queries based on previous

doi:10.14778/3675034.3675038

ones; and 5) operation timestamp pattern denotes when the encrypted database is accessed. These leakage patterns pose risks for adversaries to recover confidential queries or data through leakage attacks [9, 13, 24, 26–31, 35, 36, 41, 42, 44, 53, 81, 85].

Given its critical importance, many solutions [1, 21, 25, 38, 52, 54, 60, 88, 90] have thus been proposed to thwart these attacks by eliminating or mitigating these leakage patterns. From a performance perspective, attaining full leakage suppression for even a single pattern entails substantial overhead, some of which are even insurmountable. For instance, the well-known logarithmic lower bounds for ORAM [23, 45] almost make it theoretically infeasible to provide obliviousness in large-scale databases. And full query decorrelation with a known query distribution has been shown in [25] to be as hard as the offline ORAM. Fortunately, it is unnecessary to offer full leakage suppression in many scenarios since (1) the adversary's auxiliary information about the encrypted data and/or query workloads is practically biased or distorted, and (2) an adversary has to accumulate sufficient leakage to launch risky leakage attacks. For instance, some known-data attacks [9, 24, 35] assume explicit knowledge of a (probably large) subset of the encrypted data or queries, which seems too strong in reality. Kellaris et al. [40] show that an adversary needs $\Omega(n^4)$ range queries (exposing access patterns) to reconstruct the exact value of every record in an encrypted database of size n. And recovering data values with a relative error ϵ needs $\Omega(\epsilon^{-4})$ queries [27]. Hence, mitigating partial yet significant leakage with manageable performance overheads is destined and admissible. Considering diverse and complex deployment scenarios, such a trade-off between performance and security should ideally be tunable.

From the perspective of system leakage, most countermeasures protect only subsets of leakage patterns. For instance, ObliDB [21] provides a set of customized oblivious operators to conceal memory access patterns across various query workloads, yet it ignores hiding the sizes of the intermediate and result tables (*i.e.*, volume pattern). Most volume-hiding solutions [1, 38, 54, 88] ignore the query equality pattern, which indicates whether two queries repeat. Pancake [25] mitigates the access pattern via smoothing the access frequency to entries in an encrypted data store, while the exposed query correlation pattern has been shown to be vulnerable to IHOP attack [53]. It is hence essential to consider system-wide leakage while designing encrypted data stores. This problem is more pronounced in leakage mitigation schemes, where relaxing protection over existing leakage may inadvertently expose new and detrimental leakage patterns. A notable example is that relaxing oblivious access to frequency-smoothed access introduces the query correlation pattern [25, 53].

From the perspective of protection applied, existing solutions typically apply a uniform security notion over all supported workloads. This approach simplifies the comprehension of the system's security but impedes the attainment of fine-tuned and improved privacy-efficiency trade-offs. For instance, Adore [60] protects a set of common workloads in relational databases in a differentially oblivious approach, which ensures that the access pattern complies with the differential privacy notion. However, some workloads, such as table joins that do not preserve neighboring outputs (as discussed in §5), may not align well with such a protection measure. Besides, $\mathcal{E}$psolute [10] applies the same differentially private sanitizer to both point and range queries, while we could leverage a more efficient scheme [54] to hide the volume pattern in point queries. It is hence valuable to identify the nuanced privacy requirements inherent in each specific workload, with the primary challenge being that efficiently and securely accommodating a new workload may necessitate modifications to the existing ones.

Contributions. Drawing on the above insights, we investigate the system-wide approach towards tunable leakage mitigation by following recent enclave-based encrypted data stores [2, 21, 76, 83]. To this end, we present Swat, an efficient encrypted data store that *progressively* supports key-value, range-query, and dynamic workloads with tunable system-wide leakage mitigation. We adopt a widely accepted assumption [10, 25] that posits the existence of a *trusted client proxy*. The proxy is responsible for routing client queries to the enclave deployed on the cloud server via a secure and authenticated channel. We additionally assume a *dedicated communication channel* as the billing model based on network bandwidth (rather than data transferred) usage per month (or year), which we denote as pay-by-bandwidth, is a standard practice in cloud services. Examples in alphabetical order include Alibaba Cloud, AWS Direct Connect, Azure ExpressRoute, and Tencent Cloud.

Our work starts from the key-value workload due to its simplicity, where keys and equal-length values are protected by pseudorandom functions and authenticated encryption. We mitigate the primary leakage, *i.e.*, access and query correlation patterns, via frequency smoothing and partial query decorrelation respectively. We adopt Pancake for the former one due to its noteworthy gains in balancing performance and security. Pancake provides provable assurance of achieving a uniform access frequency for each entry in the key-value store, irrespective of their original access distribution, as long as each query is generated independently. However, it falls short in protecting encrypted data stores when queries are correlated [53]. We hence devise an elegant and almost-for-free security patch, modeled as θ-query decorrelation, to mitigate this leakage pattern. This technique also holds potential for broader applications in other searchable encryption systems.

We then extend Swat to handle range queries that inherently expose more leakage, such as order and volume patterns. To system widely address these leakage patterns, we introduce a formal security notion aimed at reducing leakage of range query systems to that of the previous stage (*i.e.*, key-value stores). Intuitively, no adversary under this notion can distinguish between a sequence of data accesses from range queries (sampled from an arbitrary distribution) and those from uniformly random sampling. To achieve this, we develop an efficient protocol that sets up the encrypted data store by partitioning the input dataset into buckets without revealing their order, and handles queries by accessing the data store at a fixed rate and retrieving a fixed number of buckets in each access. This protocol effectively suppresses order, volume, and search timestamp patterns, with reasonable monetary cost (thanks to the pay-by-bandwidth billing model).

Subsequently, we introduce a data-structure dynamization technique to enable updates over the encrypted data store. In order to maintain query efficiency, the data store requires a well-structured, albeit hidden from anyone but the trusted proxy (for security), search index. Unless properly mitigated, the memory access pattern during index updates will expose sensitive information regarding

the underlying structure of the encrypted data. We capture such a dominant privacy demand by formulating a differential obliviousness notion [14] since it offers principled privacy-efficiency trade-offs. We also evolve a k-way differentially oblivious merge algorithm from the 2-way one [14], which serves as the foundation of dynamization, to offer better privacy guarantees. Moreover, we improve the practical performance of the differentially oblivious merge algorithm by notably reducing the size of its oblivious buffer without hurting the security guarantees it claims.

We then implement an end-to-end system Swat that realizes the above functionalities on top of Intel SGX. Swat is about 10.6× slower than an encryption-only database that exposes all detrimental leakage patterns and 31.6× faster than a trivial solution that eliminates all these patterns. We also compare it to ObliDB [52], the state-of-the-art oblivious database, and $\mathcal{E}$psolute [10], the state-of-the-art range-query system mitigating the volume pattern and eliminating the memory access pattern. The result shows that our design provides competitive performance while mitigating system-wide leakage patterns. We also run extensive experiments with various settings and compiled a detailed result analysis.

We summarize our contributions in this work as follows:

- We customize a set of security models to capture varying privacy requirements in key-value, range-query, and dynamic workloads.
- We present Swat, an efficient design that *progressively* enables these workloads, offering tunable privacy-efficiency trade-offs and strategies for their systematic organization and integration.
- We implement Swat and empirically evaluate its performance over an extensive set of settings with a detailed results compilation showing the efficiency of our construction and the tunable privacy-efficiency trade-offs.

2 BACKGROUND

In this section, we first describe an outsourced data store system adapted from [10]. Then we introduce the threat model and formal security notions that capture different privacy requirements across various workloads.

2.1 Syntax and System Model

Without the loss of generality, we abstract a data store as a collection of n records r, each with a search key sk: $\mathcal{D} = \{(\mathsf{sk}_1, r_1), \ldots, (\mathsf{sk}_n, r_n)\}$. We assume that search keys take value from a *well ordered* domain $\mathcal{X} = \{1, \ldots, N\}$ for $N \in \mathbb{N}$, and all records have the same fixed bit-length. We describe the model for a single indexed attribute for ease of presentation and discuss how it can be extended to support multiple attributes.

We explicitly distinguish operations op over the data store as queries q and updates u. A query is a predicate $q : \mathcal{X} \to \{0, 1\}$ to be evaluated on $\mathcal{D}$. It results in a set $q(\mathcal{D}) = \{r_i : q(\mathsf{sk}_i) = 1\}$ containing all records whose search keys are evaluated to be true. This work focuses on the following types of queries (adapted from [10]).

Range query. A range query $q_{[x,y]}(a)$ associated with an interval $[x, y]$ is evaluated to 1 iff $x \le a \le y$. The equivalent SQL query is:

```
SELECT * FROM tab WHERE attr BETWEEN x AND y.
```

Point query. A point query $q_x(a)$ associated with an element $x \in \mathcal{X}$ is evaluated to 1 iff $x = a$. The equivalent SQL query is:

```
SELECT * FROM tab WHERE attr = x.
```

An update operation, denoted as $u = (\mathsf{updt}, \mathsf{sk}, r)$, $\mathsf{updt} \in \{\mathsf{insert}, \mathsf{delete}, \mathsf{update}\}$, performs one of the following three actions in the data store $\mathcal{D}$ that results in an updated data store $\mathcal{D}'$.

Insertion. An insertion $u = (\mathsf{insert}, \mathsf{sk}, r)$ results in $\mathcal{D}' = \mathcal{D} \cup \{(\mathsf{sk}, r)\}$. The equivalent SQL statement is:

```
INSERT INTO tab VALUES (sk, r).
```

Deletion. A deletion $u = (\mathsf{delete}, \mathsf{sk}, \perp)$ results in a (probably) new database $\mathcal{D}' \subseteq \mathcal{D}$ such that for all $(\mathsf{sk}_i, r_i) \in \mathcal{D} \setminus \mathcal{D}'$, we have $\mathsf{sk}_i = \mathsf{sk}$. The equivalent SQL statement is:

```
DELETE FROM tab WHERE attr = sk.
```

Update. An update $u = (\mathsf{update}, \mathsf{sk}, r')$ results in a new database $\mathcal{D}'$ such that for all $(\mathsf{sk}_i, r_i) \in \mathcal{D}$, we have $(\mathsf{sk}_i, r_i) \in \mathcal{D}'$ if $\mathsf{sk}_i \neq \mathsf{sk}$ and $(\mathsf{sk}_i, r') \in \mathcal{D}$ if $\mathsf{sk}_i = \mathsf{sk}$. The equivalent SQL statement is:

```
UPDATE tab SET r = r' WHERE attr = sk.
```

Outsourced dynamic data store (ODDS). An ODDS consists of three protocols between two *stateful* parties: a client C and a server $\mathcal{S}$ (adapted from [10]).

Setup protocol Π_{setup}: C takes as input a database $\mathcal{D}$ (and parameters for other purposes); $\mathcal{S}$ takes no input. C has no output (except its state); $\mathcal{S}$ outputs a data structure $\mathcal{DS}$.

Query protocol Π_{query}: C has a query q; $\mathcal{S}$ has as input $\mathcal{DS}$. C outputs $q(\mathcal{DS})$; $\mathcal{S}$ has no output. Both may update internal states.

Update protocol Π_{update}: C has an update u; $\mathcal{S}$ has input $\mathcal{DS}$. C has no formal output; $\mathcal{S}$ outputs an updated data structure $\mathcal{DS}'$. Both may update their internal states.

Correctness. We require that for any database and any operation sequence consisting of queries and updates, it holds that running Π_{setup}, and then Π_{query} and Π_{update} on the corresponding inputs, Π_{query} outputs the correct results except with negligible probability over the coins of the above runs.

Efficiency. We measure the efficiency of an ODDS from the following perspectives: 1) *Storage efficiency* measures the bit lengths of an ODDS in C and $\mathcal{S}$, including their states. The storage complexity of C should be significantly smaller than the bit length of the data store $|\mathcal{D}|$. 2) *Communication efficiency* measures the necessary network bandwidth of an ODDS rather than the bit lengths of data transferred between C and $\mathcal{S}$. We prefer the bandwidth metric as pay-by-bandwidth is a common billing model in cloud services (as discussed in §1); 3) *Search time efficiency* measures the time span between when a client issues a query and when it receives the corresponding results. 4) *Update time efficiency* measures an insertion, deletion, or updation's (amortized) processing time in $\mathcal{S}$.

2.2 Threat Model and Security Definitions

Threat model. We use Intel SGX as an example of hardware enclaves to discuss our threat model. Intel SGX offers confidentiality and integrity of data and codes inside its protected memory, *i.e.*, enclave page caches (EPC). An enclave is defined by user-level or operating system code, initiated by loading a verifiable compiled library, and interacted via well-defined functions. EPC is "uni-directly" accessible, *i.e.*, codes inside EPC can access the entire address space (except those belonging to other enclaves), but the others cannot access EPC. SGX enables a remote system to verify what code is loaded into an enclave and set up a secure communication channel with the enclave via remote attestation. Furthermore, the capacity of EPC is highly limited (*i.e.*, 128MB in total and less than 100 MB

available) compared to the untrusted memory. SGX v2 offers much richer EPC resources (up to 512 GB per processor) by dropping the integrity guarantee inside the enclave. We also take limited EPC into account while implementing Swat.

Similar to prior works [2, 21, 52, 73], we assume an *honest-but-curious adversary* with the power to continuously inspect network communication, untrusted memory, and disk, and data transferred inside the system bus (a.k.a., persistent adversary). In particular, both data flowing through the data bus and memory addresses carried in the address bus could be observed by adversaries. The latter one exposes the memory access patterns to both trusted and untrusted memory [12, 52, 86]. Adversaries can also leverage arbitrary auxiliary information (*e.g.*, a subset of encrypted data or queries, or a probably biased distribution of data records or client queries) to recover data or queries.

Timing side channels and power analysis are orthogonal to our work similar to previous studies [2, 21, 52, 73]. Denial of service attacks, which can be easily launched by a privileged attacker against an enclave, are out of scope since it does not compromise user privacy. Although there are several side-channel attacks against SGX upon speculative execution, branching history, or page faults [11, 12, 46, 84, 86], effective approaches [62, 69, 71, 72] have been proposed to mitigate these attacks and we can employ a more secure SGX implementation if necessary.

Frequency smoothing. To hide the access *distribution* of items in an encrypted key-value store, Grubbs et al. [25] proposed a security notion named "real-or-random indistinguishability under chosen (dynamic) distribution attack". Informally, it requires that no adversary is able to distinguish whether a sequence of accessed items are queried by clients or randomly sampled from a uniform distribution, *i.e.*, any characteristic distribution over encrypted items that contains sensitive information will be smoothed to a uniform one.

Pancake achieves this goal by selective replication upon initialization and batched query strategy with fake queries. Specifically, selective replication creates copies of items that are more likely to be accessed, and accesses one of them if queried. High likelihood is hence amortized to the average. It then creates fake queries for items that are less likely to be accessed, which hence raises the low likelihood to the average. Batches are introduced to ensure that fake queries are indistinguishable from real ones, and that real queries will be answered timely. The storage overhead caused by replicas could be bounded by a constant. One may also trade storage overhead α, which is defined as the total number of replicas divided by the original count, for reduced communication overhead via fewer fake queries (or vice versa). The authors use $\alpha = 2$ as default in their design and experiments.

Query decorrelation. As discussed earlier, Pancake is vulnerable when queries are correlated. Rather than a full query decorrelation notion that is as hard as offline ORAM [25], we propose the following partial query decorrelation notion, wherein we require the current query to exhibit independence from a minimum of θ previous queries. It trivially captures the decorrelation requirement since independence implies zero correlation.

Definition 1 (θ-query decorrelation). *For a discrete-time stochastic query process $\{X_t \in Q : t \geq 0\}$ where Q denotes the countable set of possible queries, we say that it satisfies θ-query decorrelation if for all $t \in \mathbb{N}_+$ and $q_0, \ldots, q_{t-1} \in Q$, there exists $S \subseteq [t]$ with $t' = |S|$ and $t' \leq \max(t - \theta, 0)$ such that $\Pr[X_t = q_t | X_0 = q_0, \ldots, X_{t-1} = q_{t-1}] = \Pr[X_t = q_t | X_{s_0} = q_{s_0}, \ldots, X_{s_{t'-1}} = q_{s_{t'-1}}]$.*

Range or random point query indistinguishability. We propose a formal security model, as shown in the full version [89], that captures the indistinguishability between data accesses from range queries (generated from a specific distribution) and data accesses from uniformly random sampling. Achieving this security goal rules out attacks based on order or volume patterns, as such leakage patterns do not exist in individual data access systems (*i.e.*, the key-value stores) with values of the same length.

Differential obliviousness (DO). DO [14, 40, 55, 56, 79, 80] essentially requires that the memory access pattern of an algorithm or data structure complies with the well-known differential privacy [19, 20] notion, which protects the privacy of individuals in published results by adding noise to the data. Since the cloud service provider is untrusted, we require access patterns in the operating the data store, rather than the outputs, to satisfy differential privacy.

Two operational sequences ops and ops′ consisting of the same number of queries are called neighboring if they differ in exactly one position i, and both are of the same update query type (insertion or deletion). Indeed, such two neighboring sequences denoted by ops $\sim$ ops′ over the same setup dataset will result in two neighboring data stores that differ in exactly one record.

Definition 2. *We say that a dynamic outsourced data store Π is (ε, δ)-differentially oblivious with respect to updates (a.k.a DO_{update}-ODDS) if for any data store $\mathcal{D}$ and any two query-consistent neighboring operational sequences* ops $\sim$ ops′*, and any possible set of memory access patterns S (adapted from [14]):*

$$\Pr[\mathcal{AP}_\Pi(\mathcal{D}, \text{ops}) \in S] \leq exp(\varepsilon) \cdot \Pr[\mathcal{AP}_\Pi(\mathcal{D}, \text{ops}') \in S] + \delta.$$

The parameter ε as a privacy loss metric adjusts the efficiency-privacy trade-off. The parameter δ allows for a negligible probability when the bound ε fails to hold, and allowing such a negligible failure probability is essential to improve system performance, as shown in [14]. The random variable $\mathcal{AP}_\Pi(\mathcal{D}, \text{ops})$ denotes the distribution of access patterns incurred by the system Π over $\mathcal{D}$ and ops.

3 SWAT DESIGN

Swat is constructed in a progressive approach that enables key-value §3.2, range-query §3.3, and dynamic §3.4 workloads with emphasis on efficiently mitigating system-wide leakage. Before delving into concrete construction, we first provide an overview of Swat, the one supporting all the above workloads, to help better understand how it works.

3.1 Overview

We provide an illustration of Swat's workflow in Fig. 1. As introduced above, Swat assumes a *trusted* proxy that routes queries from clients to the untrusted cloud server.

Workflow. Swat sets up an encrypted data store over a sorted input dataset by dividing it into buckets of the same size, randomly shuffling those buckets to hide the order pattern. Similar to Pancake [25], each bucket will be encrypted by an authenticated encryption scheme E and inserted into the backend KV store with an identifier generated by a secretly keyed pseudorandom function

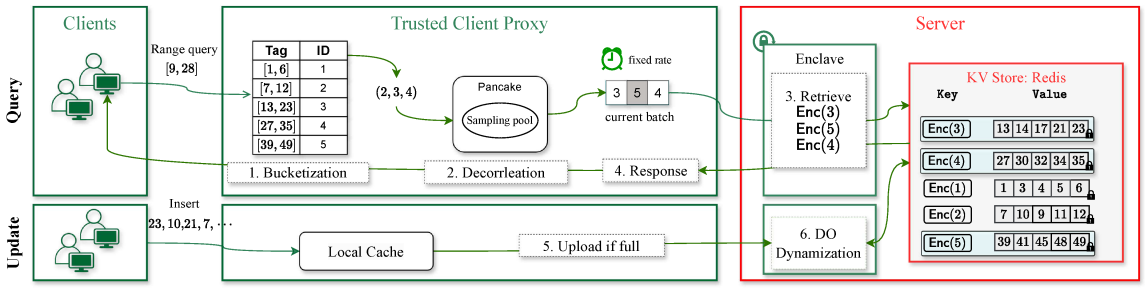

Figure 1: Upon receiving a query, the client proxy splits it into bucket accesses based on tags prepared by the cloud server's enclave. These accesses are added to a sampling pool, from which a batch of bucket IDs is randomly selected. The proxy also filters false positives in those buckets and returns the final results to the client. Upon receiving an update, the client proxy caches it until a bucket of updates accumulates. Subsequently, the client proxy sorts these updates and uploads them to the server's enclave. The enclave then updates the encrypted data store in a differentially oblivious way.

F. Upon receiving a range query from a client, the client proxy will partition it into several bucket requests according to their tags (step 1). It then puts bucket requests into a sampling pool for future processes (step 2). At each predefined time unit, it samples a batch of pending buckets as well as fake buckets and retrieves them from the backend KV store on the cloud server via a secure enclave (step 3). Once a range query receives all requested buckets, correct answers will be replied to the corresponding client by filtering the false positives (step 4). Updates over the encrypted data store are more complicated. The client proxy will maintain a small local buffer to cache newly inserted records, which will be uploaded to the cloud server as it fills up (step 5). The secure enclave in the untrusted server will then fetches necessary encrypted data, decrypt it inside its private memory, and merge several sorted record arrays into a new one in a differentially oblivious manner (step 6). Finally, the client proxy will update the necessary local states accordingly.

Parameters. We also brief system parameters to promote a high-level understanding of how they tune the performance and security of Swat. A larger θ (§3.2), which denotes the minimum number of items pending in the sampling pool, implies a stronger decorrelation effect that comes at the cost of reduced search time efficiency. A larger bucket size Z (§3.3) signifies fewer states cached in the client proxy but more false positives retrieved from the server. A large privacy loss ε as well as a smaller security parameter κ (§3.4) allows better update performance at the cost of less obliviousness. In addition, a larger k (§3.4), which denotes the number of components for dynamization, improves the update time efficiency but results in a worse search time efficiency and a larger privacy loss. Ideally, one should select a suit of parameters and conduct a benchmark test in the intended deployment environment to identify the most effective parameters, which may vary due to factors such as storage access times and network round trip times (RTTs).

3.2 θ-query Decorrelation

As outlined in §2.2, the primary leakage sources in a key-value store with a uniform value length are access frequency and query correlation patterns. The former could be efficiently fixed via the frequency smoothing technique [25]. However, this technique offers limited protection against query correlation, primarily due to its reliance on a *queue* to cache pending items for future batches. Specifically, once a query arrives, the client proxy will randomly choose a replica of the key, add it to the pending queue, and prepare a batch of accesses to the cloud server. For each access in the batch, the proxy randomly flips a coin to determine whether it is fake or real, where fake accesses are generated according to a pre-settled distribution. The proxy retrieves elements from the queue as real accesses if the queue is not empty. Otherwise, it will simulate a real query by sampling a key (replica) based on its real distribution.

The key insight here is that the *first-in-first-out* feature of the pending queue completely preserves the original query correlation. To alleviate this issue, Swat introduces a modification in Pancake by replacing the queue with a sampling pool. This adaptation allows for the random selection of pending items in the sampling pool *independently*. Consequently, correlations among *queries pending in the pool* are eliminated due to the inherent property of independence that implies zero correlation. We further introduce the following two new features to provide greater flexibility in tuning efficiency and privacy: 1) Minimum size θ that we impose on the sampling pool is an important parameter to adjust the trade-off. To meet the minimum size requirement, Swat pads the sampling pool with queries yielded by the real distribution (*i.e.*, simulating client behavior). Increasing θ can reduce query correlation, albeit with a potentially higher search time efficiency. 2) Customized sampling weight update policy allows C to dynamically adjust sampling weights over time in diverse manners. For example, weight

Algorithm 1 Sampling Pool with θ-Decorrelation in C

```
Setup(θ, π̂, fn):
  ▷ π̂: the real distribution of replicas
  ▷ fn: the update policy for weights
1: map ← ∅                          ▷ hash table
2: items ← [ ]                      ▷ item array
3: w ← [ ]                          ▷ sampling weights
4: i ← 0
5: while i < θ do
6:     r_rep ← π̂
7:     if Put(r_rep) then
8:         i ← i + 1

Put(r_rep):
1: if map.has(r_rep) then
2:     return false
3: map.insert(r_rep)
4: items.add(r_rep), w.add(1)
5: return true

Get():
1: i ← sampleIdx(items, w)
2: map.remove(items[i])
3: items.del(i), w.del(i)
4: w ← fn(w)                        ▷ update weights
5: i ← |items|
6: Execute line 5 ~ 8 of Setup
```

increasing over time allows a quasi-FIFO effect. Namely, items arriving earlier (first-in) will have a higher weight and are, therefore, more likely to be sampled in the current batch (first-out). It yields improved search time efficiency, provided that intrinsic query correlation remains within an acceptable security range.

Write support and linearizability. Pancake [25], as the base of our design, supports writes to keys in the KV store via a standard read-and-write technique in the ORAM literature [70], *i.e.*, each access consists of a read followed by a write, collectively forming a transaction. Specifically, since only sampled replicas will be written in each batch access, the proxy will maintain an UpdateCache to track which replicas of a key need to be updated in the future in the form of $k \rightarrow (v, \texttt{UpdateMap})$. UpdateMap denotes which replicas of k have been updated or need to be updated in the future. In each batch access, Pancake will consult UpdateCache to ensure that updated values propagate. Upon updating all replicas of k to $\mathcal{S}$, its corresponding entries in UpdateCache will then be removed.

In Swat, as accesses are randomly sampled from the pool, it is crucial to establish linearizability [32], *i.e.*, a database consistency guarantee ensuring that each operation appears to occur atomically and in accordance with the *real-time* ordering. We must ensure that a Get(k) on a key k, either before or after a write operation Put(k) on that key, accurately reflects the value v. Considering that Swat samples *key replicas* rather than the underlying read/write operations, we assign to each pending key k a list lst_k that tracks the *yet-to-be responded* operations to k in *real-time ordering*, exemplified by $k \rightarrow (\mathsf{read}_1, \mathsf{read}_2, \mathsf{write}_1(v_1), \mathsf{read}_3, \mathsf{write}_2(v_2))$.

Upon receiving the result of each batch access from $\mathcal{S}$, the proxy operates on each key k in the batch as follows. We denote the value received from $\mathcal{S}$ as v_0. The proxy first consults UpdateCache[k], inherited from Pancake, to verify if there is a cached update to k. If such an update exists, we replace v_0 with the cached value. The proxy then scans the list lst_k to address the pending operations as follows. For each read operation in lst_k, the proxy responds with the current value v_c. When encountering a write operation in lst_k, the proxy updates v_c with the written value and continues. Upon completing all operations in lst_k, the proxy verifies if v_c equals v_0. If this condition holds, it writes back to $\mathcal{S}$ a re-encrypted v_0. Otherwise, it overwrites the corresponding value in UpdateCache[k] with v_c and writes back to $\mathcal{S}$ an encryption of v_c. The proxy will also update UpdateMap following the approach used by Pancake.

In the above example with an initially empty UpdateCache[k], the proxy first samples a replica of k (rather than an operation on k) and obtains its value v_0 from $\mathcal{S}$. Then it responds to read_1 and read_2 with v_0, and to read_3 with v_1. After completing all operations in lst_k, the proxy will update UpdateCache[k] with v_2 and a properly configured UpdateMap. Subsequently, it will write to a replica in $\mathcal{S}$ with the encryption of v_2. In short, as Swat samples *key replicas rather than operations on keys*, and the proxy will respond to operations on keys in real-time ordering as described above, we can claim that Swat correctly establishes linearizability.

The above design is formally referred to as θ-decorrelation, as depicted in Alg. 1. The full version shows that it achieves θ-query decorrelation while adopting the unweighted sampling policy.

Efficiency. Note that the runtime complexity of Put operation is $O(1)$ since a hash table supports constant-time insertions and removals. Get operation could also be done in constant time when employing an unweighted sampling policy (*i.e.*, not weight updates) by observing that removing the i-th element from a vector could be done by two operations: 1) swapping the i-th element and the last one; 2) discarding the last element. However, if weighted sampling with dynamic weights is considered, the time complexity of Get would be $O(\theta)$ due to the possible linear scan of the weight vector. The storage overhead the sampling pool introduces is proportional to the number of pending queries, which is normally negligible compared to the local states for maintaining replica distributions.

We remark that our design is *almost for free* from the following two perspectives. In terms of implementation, it requires minimum intrusion on Pancake, as it only requires substitution of the queue with a sampling pool, both of which expose identical interfaces. Regarding performance, Swat with a uniform sampling policy ensures that Put and Get operations impose only a small constant computation overhead on the client proxy. We can hence effectively alleviate query correlation leakage with minimum efforts. Besides, we also discuss its broader interests in the full version [89].

3.3 Nearly Zero-Leakage Range Query Support

Here we present how to efficiently support nearly zero-leakage range queries based on the above design. We emphasize that even an oblivious design without appropriate padding still reveals the volume pattern, which has been demonstrated to be catastrophic under certain adversarial models [26, 30, 42]. We will go through our construction in terms of leakage patterns exposed in different stages as well as the mitigations we adopt.

Order leakage in data storage. Clearly, it is inevitable to store data in an ordered format for efficient range queries. We emphasize here that data should be stored in a way that no one but the trusted part can learn how they are ordered. Otherwise, a snapshot adversary could infer the order pattern according to the physical addresses of the data. Oblivious shuffling (via sorting with random weights) together with a local position map of the data entries could easily fix it within $O(n \log n) \sim O(n \log^2 n)$ time. In specific, the functional goal of oblivious sorting is to put the input array in order (with respect to certain keys). The obliviousness requires that the distributions of memory access patterns produced by two input arrays of the same length are indistinguishable from each other. Swat uses the classic bitonic sort algorithm due to its practical efficiency compared with other oblivious sorters (a detailed investigation can be found in the full version [89]).

Frequency and order leakage in data access. Range queries inherently exhibit characteristic data access frequencies. For instance, uniformly random range queries over the domain $\{1, \ldots, N\}$ will access the value $1 \le x \le N$ with probability $p(x) = 2x(N+1-x)/(N(N+1))$. Several query recovery attacks [27, 44] rely heavily on this frequency leakage.

Another important leakage is that data tend to be accessed sequentially without proper protection. It directly reveals the exact order among those data records. Fortunately, frequency smoothing combined with our almost-for-free query decorrelation technique allows us to mitigate such leakages effectively. For a range query $[l, r]$, we simply put all keys $x \in [l, r]$ into the sampling pool and query them in batch. Unweighted random sampling will produce a sequence of data accesses equivalent to random shuffling, which

hence fully hides the order leakage. It will, of course, introduce notable performance overhead since the result might be very sparse compared to the queried range. In addition, it cannot be directly applied to range queries over real numbers due to an infinite number of keys in between or innumerable keys for the floating type. We will address these issues shortly.

Volume, order, and timestamp leakage in data transition. Dealing with volume pattern leakage is more challenging in data transition for the following reasons: 1) Full padding implies pulling the entire database or prohibitive overhead if the maximum possible result set size is huge. 2) Padding in a differentially private way may not provide sufficient protection. 3) Replying with a subset of results introduces unacceptable false negatives in various scenarios, or replying with a sequence of subsets to avoid false positives merely coarsens the volume pattern to the subset size level.

Naïvely asking for multiple batches consecutively does not perfectly fix the issues in the third approach, as an adversary could easily infer that buckets in a short time window indeed compose a range query and are *in order*. Meanwhile, another important leakage we must consider is the search timestamp pattern. By assuming a dedicated fixed-bandwidth communication channel between C and $\mathcal{S}$ (explained in §1), we are able to eliminate the above leakage patterns simultaneously by augmenting Swat with fixed-rate bucket retrievals. We delay the formal security guarantees and first introduce the complete protocol.

To improve efficiency, we preprocess the (sorted) data store in a way that $\mathcal{D}$ will be divided into buckets of the same size Z, where each bucket will be associated with a tag denoting the range of data it contains. C will maintain tags (and other states for frequency smoothing) for buckets rather than individual search keys, resulting in significant storage space savings of $Z\times$. The only marginal modification to the query procedure, as depicted in Alg. 2, is that C has to partition a range query into a sequence of bucket requests according to the tags. The proxy will collect necessary buckets, filter false positives, and return the correct results.

Derive the distribution for each bucket being queried. After transforming the input dataset into buckets, it is necessary to estimate the bucket access distribution to smooth the frequency of bucket accesses. We adopt an assumption similar to Pancake [25] that the client proxy has an estimated distribution $\hat{\pi}$ of the true range query distribution π, while Grubbs et al. [25] also discussed how to obtain or estimate such prior knowledge on the query distribution. This means that the client proxy is aware of the probability $p_{[x,y]}$ of querying a range $[x, y]$ where x and y are within the domain $\mathcal{X}$ and $x \le y$. We also assign $p_{[x,y]}$ with 0 for $y < x$ for convenience. Then the probability of a bucket b that contains data from l to r (inclusive) being contained in a result set S is trivially given by $\Pr[b \in \mathrm{S}] = 1 - \sum_{i=1}^{l-1}\sum_{j=1}^{l-1} p_{[i,j]} - \sum_{i=r+1}^{N}\sum_{j=r+1}^{N} p_{[i,j]}$. To compute the query probability for each bucket, we derive the cumulative distribution as $P_{[x,y]} = \sum_{i=1}^{x}\sum_{j=1}^{y} p_{[i,j]}$. We could hence compute it as $\Pr[b \in \mathrm{S}] = P_{[r,N]} + P_{[N,r]} - P_{[r,r]} - P_{[l-1,l-1]}$. We also show how to efficiently compute bucket access probabilities in the context of uniform range queries, which is commonly assumed by certain attacks [27, 40, 44], in the full version.

Efficiency. Bucketize runs in $O(n \cdot \log^2(n/Z))$ time due to the oblivious shuffling process. Note that there are $O(n/Z \cdot \log^2(n/Z))$ compare-and-swap operations in shuffling, with each swap operation between two buckets taking $O(Z)$ time (*i.e.*, the bucket size). The overall running time is obtained by multiplying them together.

Partition runs in $O(\log(n/Z))$ as buckets covering the query range could be found by binary searches over $\lceil n/Z \rceil$ buckets. Line 5 of the Reply function dominates its running time due to a linear scan to filter out false positives. It runs in $O(v \cdot Z)$ where v denotes the total number of buckets it queries for. Therefore, the total query time complexity is $O(\log(n/Z) + v \cdot Z)$. We note that a larger bucket size will reduce partition time, but increase filtering time.

Algorithm 2 Bucketization

```
Bucketize(arr, Z):                                              ▷ For S:
1: n ← |arr|, B ← ⌈n/Z⌉
2: arr.add(∞, ..., ∞)                                           ▷ append B · Z − n dummies
3: Split arr into B buckets of size Z as bkt
4: tags ← [], pendingQ ← []
5: for i ← 1, ..., B do
6:     l_i ← bkt[i][1], r_i ← bkt[i][Z]
7:     tags.add([l_i, r_i]), pendingQ.add(∅)
8: w_shuffle ←$ Z^B, OSort(bkt, w_shuffle)
9: bkt', π_f, R ← Pancake.Init(π̂, bkt, α)
10: Insert bkt' into the backend KV store
11: send tags, pendingQ, π_f, R to C, who invokes Pool.Setup
```

Upon receiving a query:

```
Partition(q = [l, r]):
1: b_l ← max_{t∈tags} t.r < q.l + 1
2: b_r ← min_{t∈tags} t.l > q.r − 1
3: if b_r < b_l then return ▷ no candidate
4: q.cnt ← b_r − b_l + 1
5: q.data ← ∅
6: for i ← b_l, ..., b_r do
7:     Pool.Put(i)                                              ▷ Alg. 1
8:     pendingQ[i].add(q)
```

Upon retrieving i-th bucket:

```
Reply(i, data):
1: for q ∈ pendingQ[i] do
2:     q.data ←∪ data
3:     q.cnt ← q.cnt − 1
4:     if q.cnt ← 0 then
5:         filter q.data and send it back to the client
6: pendingQ[i] ← ∅
```

3.4 Differentially Oblivious Dynamization

There are two general approaches to enable dynamic workloads on existing Swat. The first approach is to enable the search index to accommodate newly updated data *in place*. However, such solutions suffer from either (1) limited dynamics (*e.g.*, Pancake [25], SEAL [18]) with an upper bound on the total number of data entries fixed on the setup phase. Namely, one can only insert a restricted number of entries or replace outdated entries with new ones without expanding the entire data store; or (2) expensive, *i.e.*, *super-logarithmic* overhead, such as oblivious search trees [52, 67] and differentially oblivious variants [79]. This is due to the fact that *in-place* updates require both reads to identify the appropriate positions for new values, and writes to record the new values and potentially restructure the index.

We circumvent the above predicament via the other approach, data-structure dynamization [6, 7, 49]. It refers to the process of transforming a *static* data structure into a *dynamic* one that supports arbitrarily intermixed insertions and searches. Specifically, insertions are supported by destructing old static components and

rebuilding them into a new one. The dynamization technique, called *k-binomial transform* [7], maintains k components at all times of respective sizes $\binom{D_1}{1}, \binom{D_2}{2}, \ldots, \binom{D_k}{k}$, where $0 \le D_1 < D_2 < \cdots < D_k$. The i-th component will be empty if $D_i < i$. Such a unique decomposition is guaranteed to exist. The read amplification caused by querying (at most) k components is hence independent of the data scale. We give an example in the full version [89].

Since destruction can be easily performed by retrieving bucketized data from the backend storage, the main challenge remains to design a *secure and efficient* rebuild algorithm. The functionality could be abstracted as *merging* (at most) k sorted arrays into a single sorted one. The security requires that the rebuilding process should not leak any damaging memory access pattern. We hence capture such a demand via differential obliviousness, which enjoys principled privacy-efficiency trade-offs.

Differentially oblivious merge. Chan et al. [14] proposed an (ε, δ)-differentially oblivious merge algorithm for two sorted vectors $(\mathsf{a}_0, \mathsf{a}_1)$ in $O((|\mathsf{a}_0| + |\mathsf{a}_1|)(\log \frac{1}{\varepsilon} + \log\log \frac{1}{\delta}))$ time, where neighboring inputs are two pairs of vectors $(\mathsf{a}_0, \mathsf{a}_1) \sim (\mathsf{a}'_0, \mathsf{a}'_1)$ that differ exactly in one element in total. Informally, it first allocates two arrays into two lists of bins of the same capacity Ξ in a DO manner, where each bin loads a random number of real elements along with dummies for padding. The problem is then transformed into merging two lists of sorted bins and pruning those dummies. Intuitively speaking, obliviousness comes from a small *oblivious buffer* when processing two lists, and the noisy number of elements under processing provides differentiality. See [14] for more details.

We note that the size Ξ of the oblivious buffer directly affects performance, as accessing an oblivious buffer incurs a performance overhead of $O(\log \Xi)$. Meanwhile, Ξ should be large enough to ensure a negligible probability δ that DO does not hold. Chan et al. [14] derived a theoretical lower bound on $\Xi = \Omega(\varepsilon^{-1} \log^5 \kappa)$ to guarantee $\delta = \exp(-\Theta(log^2 \kappa))$, where κ is the security parameter. In particular, we need to ensure that the sum of B i.i.d. truncated Laplace random variables parameterized by Ξ should be greater than n with all but negligible probability.

In our design, we always allocate at least $n := Z$ elements into $\hat{B} := \lceil \frac{2Z}{\Xi(1-\log^{-2}\kappa)} \rceil$ bins. Meanwhile, we can compute the distribution of the sum of $\hat{B}$ truncated Laplace random variables via the traditional convolution method. Subsequently, we can employ a binary search to determine the minimum Ξ that ensures the failure probability is less than δ. We showcase a notable improvement in reducing Ξ using this numeric method in Fig. 8.

***k*-way differentially oblivious merge.** There are two general approaches for merging k sorted arrays in a DO manner: 1) directly merge the k arrays by devising a new DO algorithm, or 2) iteratively merge two of the k arrays using the above 2-way DO merge algorithm until only a single array remains. The analysis and comparison of the two approaches are presented in the full version [89].

We choose the iterative merge as 1) the iterative merge algorithm could be easily parallelized while direct merge cannot; 2) the differential obliviousness for iterative merge could be easily obtained by the composition rule of differential obliviousness [14, 91, 92]. We notice that every element will be involved in $\log k$ instances of the DO merge algorithm, which hence results in $(\varepsilon \log k, \delta)$-DO.

Algorithm 3 Differentially Oblivious Dynamization in $\mathcal{S}$

Setup(k):
1: $D \leftarrow [0, \ldots, k-1, \infty]$ ▷ *for k-binomial transform*
2: cmpnt $\leftarrow \emptyset$

Update(records): ▷ *records in order*
1: $i \leftarrow 0, D_i \leftarrow D_i + 1$, arrs $\leftarrow$ {records}
2: **while** $D_i = D_{i+1}$ **do**
3: arrs $\leftarrow$ arrs $\cup \{\mathsf{cmpnt}_i\}$, $\mathsf{cmpnt}_i \leftarrow \emptyset$
4: $D_{i+1} \leftarrow D_{i+1} + 1, D_i \leftarrow i, i \leftarrow i + 1$
5: arrs $\leftarrow$ arrs $\cup \{\mathsf{cmpnt}_i\}$
6: arr $\leftarrow$ KWayDOMerge(arrs)
7: Bucketize(arr, Z) ▷ *Alg. 2, obtain* bkts′
8: $\mathsf{cmpnt}_i \leftarrow$ labels(bkts′) ▷ *bucket IDs*

Transform(idx, tags_{new}): ▷ *Pending query transformation in C*
For components from 1 to idx
1: **for** $t \leftarrow 1, \ldots, |\mathsf{tags}_{old}|$ **do** ▷ t *denotes* $\mathsf{tags}_{old}[t]$
2: **for** $q \in$ pendingQ$[t]$ **do**
3: Partition $([\max(q.l, \mathsf{t}.l), \min(q.r, \mathsf{t}.r)])$

However, the DO guarantee of the direct merge algorithm requires in-depth formal analysis, which complicates the situation.

We then present the differentially oblivious dynamization algorithm in Alg. 3. Swat fetches encrypted buckets in the pending components from the KV store and brings them into the enclave for decryption. It then invokes the iterative k-way DO merge algorithm to obtain a sorted array that will be *bucketized* accordingly.

We notice that Swat has to maintain structures introduced in §3.2 and §3.3 on every living component (*i.e.*, $\forall 1 \le i \le k, i \le D_i$) to ensure correctness. It leads to new issues that need to be addressed properly and securely. Firstly, we notice that buckets of small components would be accessed more frequently. For instance, if two newly inserted records with the minimum and the maximum of the domain $\mathcal{X}$ compose the only bucket in the first component, then every range query will ask for it. To alleviate such performance overhead, the client proxy maintains a local cache of size Z (*i.e.*, the bucket size). $\mathcal{C}$ sends a bucket of sorted records together to $\mathcal{S}$ only if the local cache is full.

Secondly, we need to guarantee the correctness of queries in components to be destroyed that have not yet received responses from all pending buckets. Meanwhile, we should minimize the number of targeted buckets in the new component, aiming to introduce minimal performance overhead. A naïve solution is to append pending queries of an old bucket with tag t_{old} to newly generated buckets with tags tags by invoking Partition(t_{old}) in Alg. 2, *i.e.*, treating the old tag as a query. However, such a solution may result in many unnecessary bucket retrievals. In the instance above, a bucket with tag $[\min, \max]$ will propagate its pending queries to *all* buckets in the newly generated component, which will download the entire data store if it happens to be the k-th component. Fortunately, this issue could be easily fixed by invoking Partition in the interval $([\max(q.l, \mathsf{t}_{old}.l), \min(q.r, \mathsf{t}_{old}.r)])$, where q is the query pending for future replies. We name it pending query transformation and provide a formal description in Alg. 3.

We show its formal security in the full version [89]. Informally, it satisfies $(\varepsilon \log k, \delta)$-$\mathrm{DO}_{\mathrm{update}}$-ODDS, wherein the argument follows a k-fold sequential composition of DO [91]. We remark that Swat sustains a consistent level of *per-insertion* privacy loss, irrespective

of the cumulative number of insertions into the encrypted data store. This contrasts to the log-structured merge tree employed in [14], which offers $(\varepsilon \log |\text{ops}|, \delta)$-DO and experiences a *per-insertion* privacy loss that escalates with the cumulative number of inertions performed. Furthermore, we can apply a standard sequential composition rule for DO [91] to compute the cumulative privacy loss of all updates to the encrypted data store.

Update and delete supports. Updates are performed by inserting new record values with more recent operating timestamps. Deletion is indeed a special update with a special record type known as a tombstone. Therefore, the client proxy needs to sort the query results according to the search keys and cancel outdated records according to the operating timestamps. To further reduce performance overhead, we could mark all outdated records as dummies that will be purged while merging these components.

3.5 Security Assurance and Efficiency Tuning

Following the description of Swat design, we proceed to review its security assurance and discuss how to tune efficiency with security. As Swat evolves to support key-value searches, static range queries, and dynamic range queries, we will also discuss its security considerations in each of these scenarios. We emphasize that, throughout our design, we do not consider publishing or sharing private query results, but rather focus on preventing the untrusted server from recovering private client queries via various leakage patterns.

θ-query decorrelation in frequency-smoothed KV stores. Swat aims to mitigate query correlation leakage in frequency-smoothed key-value stores through random sampling. It mitigates the *overall* query correlation by eliminating the correlation among *pending* queries. This, however, comes at the cost of introducing unstable or increased query latency for users, particularly evident when handling numerous pending queries or enforcing a minimum size on the sampling pool. The performance overhead is trivial, as the expected times for a query to be sampled uniformly at random from θ pending queries is θ. For instance, Swat can set $\theta = n$ to align the size of the sampling pool with that of the data store to maximize security. Consequently, the trusted client proxy will *randomly* retrieve an object from the untrusted server over the entire data store in each access. This ensures that server observations of data accesses remain entirely *unrelated* to the client's real queries. However, it results in severely restricted utility, since retrieving the target query requires, on average, n random requests to the server. We also empirically show that $\theta \in [2, 4]$ is adequate to mitigate query correlation leakage in a classic Markov query process.

Nearly zero-leakage static range queries. Swat then advances to support *static* range queries, offering a strong security guarantee that ensures requests involving range queries to the untrusted server remain indistinguishable from point queries. Namely, given a sequence of requests, adversaries cannot discern if certain queries constitute a range query. And it relies on an additional yet reasonable assumption of the presence of a stable communication channel, allowing the trusted proxy to consistently query the untrusted server to conceal sensitive information. A crucial parameter in this scenario, the bucket size Z, affects only the query efficiency. And its optimization, as discussed in §3.3, involves a delicate balance between partition time and filtering time, depending on the specific characteristics of the workload.

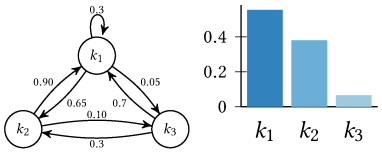

Figure 2: Markov model (left) and its stationary distribution of queried keywords (right).

Differentially oblivious updates. Subsequently, Swat expands its functionalities to handle *updates* to the underlying data store. We remark that to maintain an efficient search index, Swat has to arrange the updated data within the index, rather than simply appending it to the end. However, this "rearrangement" process may inadvertently reveal sensitive access pattern leakage. Therefore, Swat introduces differential obliviousness [14] (DO) to protect updated values from such leakage. Unlike DP, a notion protecting the *values* of individuals, DO employs analogous principles but protects the *access pattern* introduced by individuals, particularly through updates in our scenario. Furthermore, we emphasize that the access pattern reveals sensitive but incomplete information about updated data, and adversaries need additional auxiliary information to compromise client privacy [35, 37, 53]. Therefore, one may not need a considerably small ε to mitigate such leakage. In cases where clients do require such a small ε, the efficiency of Swat will degrade to an oblivious one, ensuring that no sensitive information about updated data will be leaked. Conversely, when aiming for maximal efficiency with a large ε, Swat will operate similarly to a nonoblivious one, disclosing the locations of updated data and potentially revealing their sensitive values.

4 EXPERIMENTAL EVALUATION

We implemented Swat as a modular client-server application on Intel SGX [16] in C++. Our implementation builds upon the Remote Attestation sample code provided with the SGX SDK [34] and utilizes its libraries for encryption, MACs, and hashing.

4.1 Experimental Setup

Environment. We run experiments on a machine with an Intel(R) Xeon(R) Platinum 8369B CPU @ 2.90GHz of 32 physical cores, with SGXv2 enabled. The machine has 128GB RAM, of which about 64GB is enclaves' protected memory. It operates on Ubuntu 20.04 and uses SGX SDK version 2.19.

Dataset and query. We use synthetic datasets with varying dataset sizes, record lengths, and domain sizes. We also generate several query sets with different selectivities (*i.e.*, lengths of the range). Queries are sampled uniformly from the relevant domain.

Default parameters. Unless otherwise specified, a number of parameters are set to default values. The default privacy parameters are $\varepsilon = 1.0$, $\lambda = 512$ (*i.e.*, $\delta = e^{-\log^2 \lambda} \approx 10^{-35}$), $\theta = 5$ and storage overhead $\alpha = 2$ for frequency smoothing. The default sampling strategy is to sample with equal and constant weights. We choose a default bucket capacity $Z = 512$. The default data store consists of $n = 10^6$ uniformly sampled records of size 4 KB from the domain

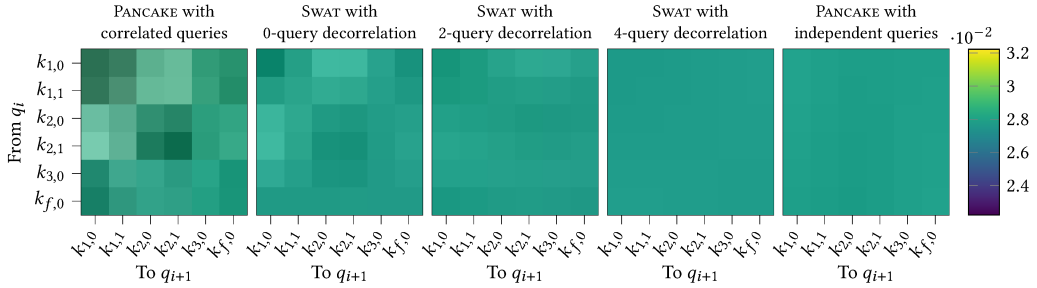

Figure 3: Observed query transition frequencies following the Markov process in Fig. 2. Reduced color variation signifies a decrease in observed query correlation. The rightmost figure represents a scenario in which queries are generated independently.

$[1, 10^6]$. Queries are uniformly sampled with selectivity $\sigma = 0.5\%$, and the optimal batch size for frequency smoothing is estimated as $\lceil 3 \cdot n\sigma/Z \rceil$ according to Grubbs et al. [25]. We use Redis [64] as the default backend key-value store, and Swat is instantiated with $k = 8$ and parallelized with 32 threads (for k-way DO merge and bitonic sorting). And the effective bandwidth is 100 Mbps.

Compared approaches. We set up two sets of experiments for comparison with the baseline approaches: one in the context of key-value stores, aligning with Pancake's specialization (corresponding to Swat in §3.2), and the other in a dynamic range query scenario.

In the first setting, we focus on the efficacy of query decorrelation and the performance overhead introduced by Swat relative to Pancake. For a clearer demonstration of query decorrelation efficacy, we adopt a simple and classic assumption that client queries follow a Markov process, *i.e.*, the current query depends only on the previous one. An adversary can then recover encrypted queries by analyzing the transition frequencies among those queries and the (esitmated) Markov transition probabilities [53].

In light of the above, achieving uniform or smoothed transition frequencies helps in thwarting such attacks, as *uniform* transition frequencies indicate no query correlation. We will hence measure the efficacy of query decorrelation via two approaches: 1) heat maps, where different colors denote different transition frequencies, and reduced color variation hence signifies a decrease in observed query correlation; and 2) relative standard deviation (RSD), quantifying the degree of dispersion in observed frequencies relative to the uniform, where a smaller RSD implies less observed query correlation.

As outlined in §3.5, the primary efficiency overhead introduced by Swat over Pancake is its variable or increased latency, measured in batches required to retrieve the target object. Additionally, as Pancake duplicates n key-value pairs into $2n$ replicas, we limit n to 3 for a clearer presentation in the heat maps. The chosen Markov model and the stationary distribution of keys are shown in Fig. 2.

We also benchmark the full version of Swat against the following approaches with security ranked from weak to strong.

Encryption-only encrypted databases. Current commercial products (*e.g.*, Azure Always-encrypted [2], Operon [83]) ignoring leakage mitigation represent the most efficient but least private and secure solution. We take StealthDB [76] as an example.

ObliDB. It represents oblivious solutions that fully eliminate access pattern but ignore the volume pattern [21]. We estimate the query latency via the StealthDB without index as ObliDB linearly scans the encrypted data store at least twice to answer range queries.

$\mathcal{E}$psolute. $\mathcal{E}$psolute combines differential privacy and ORAM to mitigate the volume pattern and fully conceal the memory access pattern, respectively. Upon receiving a range query, the user (or a trusted client proxy C similar to our setting) identifies the true IDs of records targeted by the query via the local indices, and the server $\mathcal{S}$ uses a DP sanitizer to compute the noisy number of records c for C. C then prepares fake IDs based on c and sends both true and fake IDs to $\mathcal{S}$ to retrieve corresponding records through an ORAM protocol. The above solution could still be prohibitively slow in practice. Fortunately, the authors bring it to real-world requirements by initiating parallel local indices and ORAMs. Similar design ideas are also present in ShortStack [78], a distributed and optimized Pancake design. Note that we can also draw upon such concepts to scale Swat horizontally. Therefore, we limit $\mathcal{E}$psolute with a single ORAM for fairness.

Swat. We place Swat here to mark its more comprehensive security guarantees in comparison to the above three baselines.

Linear scan. It serves as the most private and secure albeit inefficient baseline that C will download the entire encrypted data store from $\mathcal{S}$, decrypt, and linearly scan to answer every query. It eliminates all five detriment leakage patterns on which we focus. We further enable parallel download for a fair comparison with Swat.

4.2 Experiment Stages

Query decorrelation and performance compared to Pancake. As shown in Fig. 3, the direct application of our method, even without the minimum requirement on the size of the sampling pool (*i.e.*, $\theta = 0$), will significantly smooth the transition frequencies compared to those of the Pancake design with correlated queries. When $\theta = 4$, the transition frequencies of our proposed approach are close to the uniform ones, *i.e.*, independent queries with no correlation. The results imply that Swat does effectively decorrelate client queries, thereby thwarting frequency analysis attacks [36, 53].

To show the tunable trade-offs between privacy and efficiency of Swat, we evaluate Swat across different sampling pool sizes θ and weight update policies. Specifically, we employ the relative standard deviation (RSD) metric to quantify privacy benefits and measure the efficiency of Swat in terms of batches required to retrieve the target data as discussed in §4.1. Figure 4 shows the RSD and latency of Swat and Pancake with different settings.

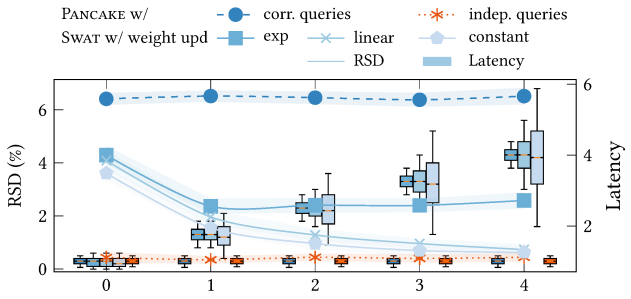

Figure 4: Privacy benefits and efficiency penalties over varying sampling pool sizes θ and weight update policies against Pancake. RSD lines denotes relative standard deviation that measures the degree of dispersion of frequencies relative to the uniform. Latency bars, measured by the number of batches to retrieve the target data, serves as a key performance indicator to reflect system responsiveness. The three middle bars from left to right represent exponential, linear, and constant updates to sampling weights in each batch.

Table 1: Storage usage in default setting.

	C MB	S GB
StealthDB	0	3.81
$\mathcal{E}$psolute	28.9	12.0
Swat	0.18	7.65

Table 2: Bandwidth costs (MB) with various σ and n.

σ \ n	10^5	10^6	10^7
0.1%	6	12	118
0.2%	6	24	236
0.5%	8	58	586

In terms of privacy, we find that when Swat updates the sampling weight linearly or constantly, the relative standard deviation decreases rapidly and becomes comparable to that of the independent case at $\theta = 4$, showing better privacy benefits. On the other hand, when the sampling weights are updated exponentially, the RSD stabilizes after dropping to 4%. This is because, in this case, the sampling pool behaves akin to a FIFO queue, where correlation among pending queries is more likely to be preserved. Nevertheless, Swat still outperforms Pancake, whose RSD is greater than 6%.

It is also evident that the aforementioned privacy benefits come at the expense of unstable or increased latency. The average latency increases from 1.7× to 4× as θ increases from 1 to 4, consistent with our analysis in §3.5. We also notice that the faster the sampling weights grow over time, the more closely the distribution of latency behaves to a FIFO queue, which also confirms our viewpoint.

We will then move on to the evaluation of the full version of Swat.

Storage and bandwidth costs. The storage cost of two baseline approaches and Swat is presented in Table 1. It shows that C only needs to maintain a *tiny* local state that is 161× smaller than $\mathcal{E}$psolute. Meanwhile, the server storage is approximately twice that of the encryption-only one, and is more competitive than $\mathcal{E}$psolute.

Query bandwidth costs in Swat are also reasonable as shown in Table 2. In specific, a dedicated 1 Gbps network channel can support querying less than 0.1% of the data from a database with 10 million 4 KB records (equivalent to 40 GB) per second.

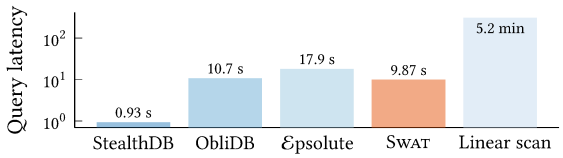

Figure 5: *Range-query* systems under the default setting. The security strength increases from left to right.

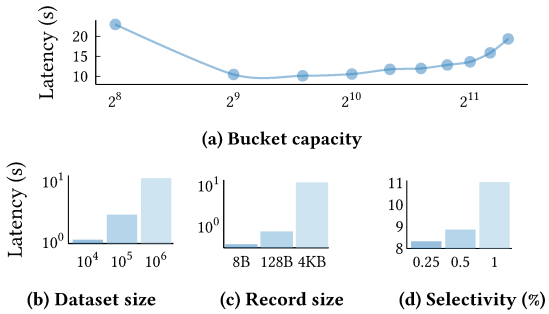

Figure 6: Query latency under different settings.

Performance against baseline approaches. We run experiments with default parameters on Swat and the aforementioned baselines (Fig. 5). StealthDB, configured for (approximately) maximum performance and no leakage mitigation, completes queries within one second, which is just 10.6× faster than Swat. It also implies the necessary time costs associated with encryption/decryption, enclave operations, and network transmission. Linear scan illustrates the efficiency and practicality of Swat compared to the trivially zero-leakage solution. The difference is 31.6× for the default setting. In addition, the query latency is competitive against the other two baselines that hide only *subsets of* the leakage patterns we identified. Also, $\mathcal{E}$psolute lacks support for dynamic workloads.

Performance with varying parameters. We have varied many configuration parameters to measure and understand the impact of them on the performance of Swat, including the search time efficiency (in Fig. 6) and the update time efficiency (in Fig. 7).

Bucket capacity (Fig. 6a). We observe a notable decrease in query latency from a small Z to a moderate one, while the latency gradually increases for larger Z. A small bucket size performs worse than moderate sizes due to 1) more buckets the dataset being partitioned into and hence more time to locate target buckets, and 2) more batches to fetch the same amount of data. Conversely, a large bucket size leads to more false positives being transmitted and filtered by the client proxy, thereby resulting in higher latency. For instance, increasing the bucket size from 2^8 to 2^9 reduces the (expected) number of batches (*i.e.*, roundtrips) required to fulfill a range query from 4 to 2, thus effectively reducing query latency. While further increasing the bucket size to 2^{11} increases false positive rates from 24.8% to 39.0%, hence increasing filtering time.

Data, record and result size (Figs. 6b to 6d). We observe query overheads against all three parameters to be positive but nonlinear, due to discrete batch sizes and bucket sampling.

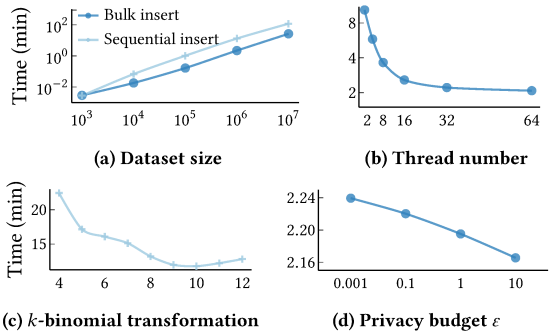

Figure 7: Inserting records under different settings. Records are inserted all at once in bulk mode and bucket by bucket in sequential mode.

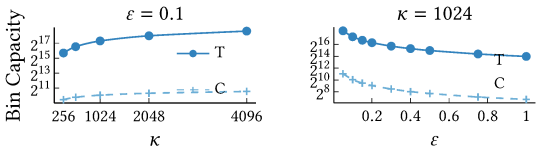

Figure 8: Minimal bin capacity obtained through the theoretical analysis (T) [14] vs. convolutional computation (C) with different security parameters κ and privacy budgets ε.

Data size and thread num on updates (Figs. 7a and 7b). We have tried 10^3 to 10^7 data insertions in both bulk and sequential modes. It is evident that the former mode outperforms the sequential one by orders due to computation saves on the dynamization process. Besides, the *logarithmized* running time of both modes grows linearly with the logarithmized data size, matching our asymptotic analysis.
Component number (Fig. 7c). The total running time for one million sequential insertions drops as the value k increases at the beginning and gradually increases when k exceeds 10. It conforms to the theoretical write overhead $(k!n)^{1/k}$ that obtains its minimum at $k = 10$ when $n = 10^6$ (for $k \in \mathbb{N}$).
Privacy budget ε (Fig. 7d). ε determines the oblivious buffer size and hence affects the setup time. A stricter privacy budget demands a larger oblivious buffer, resulting in a longer running time.

5 RELATED WORK

CryptDB [58] and the subsequent MONOMI [75] utilize special cryptographic primitives to allow different operations over ciphertexts with different security levels. Arx [57] improves the security level by using semantically secure encryption only. There are numerous differential privacy data analytics systems [15, 22, 43, 51]. This line of work assumes a fully trusted data curator (*i.e.*, the cloud service provider) that deviates greatly from our security model. Meanwhile, encrypted databases based on secure multiparty computation and homomorphic encryption are also evolving. However, the former designs [5, 47, 77, 82, 87] require multiple noncolluding servers (which also deviate from our setting), and the latter ones [8, 63, 66, 68, 74] are yet to be practically deployed on large-scale datasets. Another line of work we follow leverages hardware enclaves, such as Cipherbase [3], TrustedDB [4], StealthDB [76], EnclaveDB [59], Enclage [73], Operon [83].

There is also a line of work focusing on leakage suppression in encrypted databases. Opaque [90], Oblix [52], Obladi [17], POSUP [33], HIRB [67], and ObliDB [21] focus on fully hiding access pattern via different oblivious designs. Given significant overhead, recent work has increasingly concentrated on leakage mitigation that also maintains acceptable performance. Frequency smoothing [25, 50, 78] does not hide which data are accessed but the frequency to be accessed. Maiyya et al. [48] propose a novel security notion to measure the uniformity of accesses to *key-value stores*. They also developed Waffle [48] that achieve this notion and mitigate both access frequency and query correlation patterns without prior knowledge of data access distribution, offering tunable security-performance trade-offs as well. Differential obliviousness [14, 40, 55, 56, 79, 80] as another promising direction introduces the differential privacy notion into the protection of memory access patterns. Similarly, many works mitigate volume pattern leakage by adding differentially private noise to the result set. Coarse-grained obliviousness [61, 65, 73] relaxes the requirement for indistinguishable distributions of memory blocks to higher granularity (*e.g.*, memory pages). And coarse-grained volume-hiding techniques [18, 39, 52] follow a similar approach by retrieving results in batches , trading correctness for security by returning a fixed number of results per query, or adjustably padding the result set to its nearest power of a user-defined parameter. In addition, Wang et al. [81] considered the update timestamp pattern and mitigated this by updating at a fixed rate or updating only if the number of updates satisfies a noised threshold that satisfies differential privacy.

6 CONCLUSION

Systematic leakage mitigation in encrypted data stores has become a critical problem. In this paper, we present Swat, an enclave-assisted encrypted data store that efficiently mitigates various leakages in key-value, range-query, and dynamic workloads. We decorrelate queries with an almost-for-free sampling pool to enhance the security of Pancake. We implement a system prototype and conduct a comprehensive evaluation on extensive and diverse datasets and workloads, demonstrating excellent performance while meeting the specified security definitions.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their insightful feedback. This work is supported in part by the National Key Research and Development Program of China 2023YFB2904000, by the National Natural Science Foundation of China under Grant (NSFC) 62202228, U20A20178, U23A20306, 62072395, 62032021, 62206207, by the Natural Science Foundation of Jiangsu Province under Grant BK20210330, by the Fundamental Research Funds for the Central Universities 30923011023, and by HK RGC under Grants CityU 11217620, RFS2122-1S04, R6021-20F, R1012-21, C2004-21G, and C1029-22G. This work is also partially supported by Alibaba Group through Alibaba Innovative Research Program.

REFERENCES

[1] Ghous Amjad, Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2023. Dynamic Volume-Hiding Encrypted Multi-Maps with Applications to Searchable Encryption. *Proc. of PETs* (2023).

[2] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. 2020. Azure SQL Database Always Encrypted. In *Proc. of ACM SIGMOD*.

[3] Arvind Arasu, Spyros Blanas, Ken Eguro, Manas Joglekar, Raghav Kaushik, Donald Kossmann, Ravishankar Ramamurthy, Prasang Upadhyaya, and Ramarathnam Venkatesan. 2013. Secure database-as-a-service with Cipherbase. In *Proc. of ACM SIGMOD*.

[4] Sumeet Bajaj and Radu Sion. 2014. TrustedDB: A Trusted Hardware-Based Database with Privacy and Data Confidentiality. *IEEE Transactions on Knowledge and Data Engineering* 26, 3 (2014), 752–765.

[5] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. ShrinkWrap: Efficient SQL Query Processing in Differentially Private Data Federations. *Proc. VLDB Endow.* 12, 3 (2018), 307–320. https://doi.org/10.14778/3291264.3291274

[6] Jon Louis Bentley. 1979. Decomposable Searching Problems. *Inf. Process. Lett.* 8, 5 (1979), 244–251.

[7] Jon Louis Bentley and James B. Saxe. 1980. Decomposable Searching Problems I: Static-to-Dynamic Transformation. *J. Algorithms* 1 (1980), 301–358.

[8] Song Bian, Zhou Zhang, Haowen Pan, Ran Mao, Zian Zhao, Yier Jin, and Zhenyu Guan. 2023. HE3DB: An Efficient and Elastic Encrypted Database Via Arithmetic-And-Logic Fully Homomorphic Encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2930–2944.

[9] Laura Blackstone, Seny Kamara, and Tarik Moataz. 2020. Revisiting Leakage Abuse Attacks. In *Proc. of NDSS*.

[10] Dmytro Bogatov, Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. 2021. $\mathcal{E}$psolute: Efficiently Querying Databases While Providing Differential Privacy. In *Proc. of ACM CCS*.

[11] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proc. of WOOT*.

[12] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proc. of USENIX Security*.

[13] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. 2016. Leakage-Abuse Attacks Against Searchable Encryption. *IACR Cryptology ePrint Archive* (2016), 718.

[14] T.-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. 2019. Foundations of Differentially Oblivious Algorithms. In *Proc. of SODA*.

[15] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. 2020. Cryptε: Crypto-Assisted Differential Privacy on Untrusted Servers. In *Proc. of ACM SIGMOD*.

[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* (2016), 86.

[17] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *Proc. of OSDI*.

[18] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. 2020. SEAL: Attack Mitigation for Encrypted Databases via Adjustable Leakage. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2433–2450.

[19] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Proc. of TCC*.

[20] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Found. Trends Theor. Comput. Sci.* 9, 3-4 (2014), 211–407.

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

[22] Chang Ge, Xi He, Ihab F. Ilyas, and Ashwin Machanavajjhala. 2019. APEx: Accuracy-Aware Differentially Private Data Exploration. In *Proc. of ACM SIGMOD*.

[23] Oded Goldreich. 1987. Towards a theory of software protection and simulation by oblivious RAMs. *Proc. of ACM STOC* (1987).

[24] R. Groot Roessink. 2020. Experimental review of the IKK query recovery attack: Assumptions, recovery rate and improvements.

[25] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. 2020. Pancake: Frequency Smoothing for Encrypted Data Stores. In *Proc. of USENIX Security*.

[26] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Pump up the Volume: Practical Database Reconstruction from Volume Leakage on Range Queries. In *Proc. of ACM CCS*.

[27] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2019. Learning to Reconstruct: Statistical Learning Theory and Encrypted Database Attacks. In *Proc. of IEEE S&P*. 1067–1083.

[28] Paul Grubbs, Richard McPherson, Muhammad Naveed, Thomas Ristenpart, and Vitaly Shmatikov. 2016. Breaking Web Applications Built On Top of Encrypted Data. In *Proc. of ACM CCS*.

[29] Paul Grubbs, Kevin Sekniqi, Vincent Bindschaedler, Muhammad Naveed, and Thomas Ristenpart. 2017. Leakage-Abuse Attacks against Order-Revealing Encryption. In *Proc. of S&P*.

[30] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. 2019. Encrypted Databases: New Volume Attacks against Range Queries. In *Proc. of ACM CCS*.

[31] Zichen Gui, Kenneth G. Paterson, and Tianxin Tang. 2023. Security Analysis of MongoDB Queryable Encryption. In *Proc. of USENIX Security*.

[32] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492. https://doi.org/10.1145/78969.78972

[33] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A. Yavuz. 2019. Hardware-Supported ORAM in Effect: Practical Oblivious Search and Update on Very Large Dataset. *Proc. of PETs* (2019).

[34] Intel. 2020. Intel/linux-sgx. Online at https://github.com/intel/linux-sgx.

[35] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. 2012. Access Pattern disclosure on Searchable Encryption: Ramification, Attack and Mitigation. In *Proc. of NDSS*.

[36] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Jamie DeMaria, Andrew Park, and Amos Treiber. 2023. MAPLE: MArkov Process Leakage attacks on Encrypted Search. Cryptology ePrint Archive, Paper 2023/810. https://eprint.iacr.org/2023/810 https://eprint.iacr.org/2023/810.

[37] Seny Kamara, Abdelkarim Kati, Tarik Moataz, Thomas Schneider, Amos Treiber, and Michael Yonli. 2022. SoK: Cryptanalysis of Encrypted Search with LEAKER - A framework for LEakage AttacK Evaluation on Real-world data. In *Proc. of IEEE EuroS&P*.

[38] Seny Kamara and Tarik Moataz. 2019. Computationally Volume-Hiding Structured Encryption. In *Proc. of EUROCRYPT*.

[39] Seny Kamara, Tarik Moataz, and Olga Ohrimenko. 2018. Structured Encryption and Leakage Suppression. In *Proc. of CRYPTO (Lecture Notes in Computer Science)*.

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

[41] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2019. Data Recovery on Encrypted Databases With k-Nearest Neighbor Query Leakage. In *Proc. of S&P*.

[42] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. 2021. Response-Hiding Encrypted Ranges: Revisiting Security via Parametrized Leakage-Abuse Attacks. In *Proc. of IEEE S&P*.

[43] Ios Kotsogiannis, Yuchao Tao, Xi He, Maryam Fanaeepour, Ashwin Machanavajjhala, Michael Hay, and Gerome Miklau. 2019. PrivateSQL: A Differentially Private SQL Query Engine. *Proc. VLDB Endow.* (2019).

[44] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. 2018. Improved Reconstruction Attacks on Encrypted Data Using Range Query Leakage. In *Proc. of IEEE S&P*.

[45] Kasper Green Larsen and Jesper Buus Nielsen. 2018. Yes, There is an Oblivious RAM Lower Bound!. In *Advances in Cryptology – CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II* (Santa Barbara, CA, USA). Springer-Verlag, Berlin, Heidelberg, 523–542. https://doi.org/10.1007/978-3-319-96881-0_18

[46] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Proc. of USENIX Security*.

[47] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. SECRECY: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1031–1056. https://www.usenix.org/conference/nsdi23/presentation/liagouris

[48] Sujaya Maiyya, Sharath Chandra Vemula, Divyakant Agrawal, Amr El Abbadi, and Florian Kerschbaum. 2023. Waffle: An Online Oblivious Datastore for Protecting Data Access Patterns. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–25.

[49] Claire Mathieu, Rajmohan Rajaraman, Neal E. Young, and Arman Yousefi. 2021. Competitive Data-Structure Dynamization. In *Proc. of SODA*, Dániel Marx (Ed.).

[50] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. 2015. Modular Order-Preserving Encryption, Revisited. In *Proc. of ACM SIGMOD*.

[51] Frank McSherry. 2010. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. *Commun. ACM* 53, 9 (2010), 89–97.

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

[53] Simon Oya and Florian Kerschbaum. 2022. IHOP: Improved Statistical Query Recovery against Searchable Symmetric Encryption through Quadratic Optimization. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 2407–2424.

[54] 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 *Proc. of ACM CCS*.

[55] Giuseppe Persiano and Kevin Yeo. 2019. Lower Bounds for Differentially Private RAMs. In *Proc. of EUROCRYPT*, Yuval Ishai and Vincent Rijmen (Eds.).

[56] Giuseppe Persiano and Kevin Yeo. 2022. Lower Bound Framework for Differentially Private and Oblivious Data Structures. Cryptology ePrint Archive, Paper 2022/1553.

[57] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. 2019. Arx: An Encrypted Database Using Semantically Secure Encryption. *Proc. of VLDB* 12, 11 (jul 2019), 1664–1678.

[58] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proc. of ACM SOSP*.

[59] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *Proc. of IEEE S&P*.

[60] Lianke Qin, Rajesh Jayaram, Elaine Shi, Zhao Song, Danyang Zhuo, and Shumo Chu. 2022. Adore: Differentially Oblivious Relational Database Operators. *Proc. VLDB Endow.* 16, 4 (2022), 842–855.

[61] Maan Haj Rachid, Ryan D. Riley, and Qutaibah M. Malluhi. 2020. Enclave-based oblivious RAM using Intel's SGX. *Comput. Secur.* 91 (2020), 101711. https://doi.org/10.1016/j.cose.2019.101711

[62] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *In Proc. of USENIX Security*.

[63] Brandon Reagen, Wooseok Choi, Yeongil Ko, Vincent T. Lee, Hsien-Hsin S. Lee, Gu-Yeon Wei, and David Brooks. 2021. Cheetah: Optimizing and Accelerating Homomorphic Encryption for Private Inference. In *Proc. of IEEE HPCA*.

[64] Redis. 2022. https://redis.io/.

[65] Kui Ren, Yu Guo, Jiaqi Li, Xiaohua Jia, Cong Wang, Yajin Zhou, Sheng Wang, Ning Cao, and Feifei Li. 2020. HybrIDX: New Hybrid Index for Volume-hiding Range Queries in Data Outsourcing Services. In *Proc. of IEEE ICDCS*.

[66] Xuanle Ren, Le Su, Zhen Gu, Sheng Wang, Feifei Li, Yuan Xie, Song Bian, Chao Li, and Fan Zhang. 2022. HEDA: Multi-Attribute Unbounded Aggregation over Homomorphically Encrypted Database. *Proc. VLDB Endow.* (2022).

[67] Daniel S. Roche, Adam J. Aviv, and Seung Geol Choi. 2016. A Practical Oblivious Map Data Structure with Secure Deletion and History Independence. In *Proc. of IEEE S&P*.

[68] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. 2021. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. In *In Proc. of MICRO*.

[69] Jaebaek Seo, Byoungyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *NDSS*.

[70] Elaine Shi. 2020. Path Oblivious Heap: Optimal and Practical Oblivious Priority Queue. In *Proc. of IEEE S&P*.

[71] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *In Proc. of NDSS*.

[72] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and P. Saxena. 2016. Preventing Page Faults from Telling Your Secrets. *Proc. of ACM AsiaCCS* (2016).

[73] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building Enclave-Native Storage Engines for Practical Encrypted Databases. *Proc. of VLDB* (2021).

[74] Benjamin Hong Meng Tan, Hyung Tae Lee, Huaxiong Wang, Shu Qin Ren, and Khin Mi Mi Aung. 2021. Efficient Private Comparison Queries Over Encrypted Databases Using Fully Homomorphic Encryption With Finite Fields. *IEEE Trans. Dependable Secur. Comput.* 18, 6 (2021), 2861–2874.

[75] Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing Analytical Queries over Encrypted Data. *Proc. VLDB Endow.* 6, 5 (2013), 289–300. https://doi.org/10.14778/2535573.2488336

[76] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proc. of PETs* 2019, 3 (2019), 370–388.

[77] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–18.

[78] Midhul Vuppalapati, Kushal Babel, Anurag Khandelwal, and Rachit Agarwal. 2022. SHORTSTACK : Distributed, Fault-tolerant, Oblivious Data Access. Cryptology ePrint Archive, Paper 2022/662. https://eprint.iacr.org/2022/662 https://eprint.iacr.org/2022/662.

[79] Sameer Wagh, Paul Cuff, and Prateek Mittal. 2016. Root ORAM: A Tunable Differentially Private Oblivious RAM. *CoRR* abs/1601.03378 (2016). arXiv:1601.03378

[80] Sameer Wagh, Paul Cuff, and Prateek Mittal. 2018. Differentially private oblivious ram. *Proc. of PETs* 2018, 4 (2018), 64–84.

[81] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2021. DP-Sync: Hiding Update Patterns in Secure Outsourced Databases with Differential Privacy. In *Proc. of ACM SIGMOD*.

[82] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2022. IncShrink: architecting efficient outsourced databases using incremental mpc and differential privacy. In *Proceedings of the 2022 International Conference on Management of Data*. 818–832.

[83] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, Xuntao Cheng, Xiaolong Xie, and Yu Zou. 2022. Operon: An Encrypted Database for Ownership-Preserving Data Management. In *Proc. of VLDB*.

[84] Nico Weichbrodt, Anil Kurmus, Peter R. Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *Proc. of ESORICS*.

[85] Lei Xu, Leqian Zheng, Chengzhi Xu, Xingliang Yuan, and Cong Wang. 2023. Leakage-Abuse Attacks Against Forward and Backward Private Searchable Symmetric Encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 3003–3017.

[86] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proc. of S&P*.

[87] Yanping Zhang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2023. Longshot: Indexing Growing Databases using MPC and Differential Privacy. *Proc. VLDB Endow.* 16, 8 (2023), 2005–2018. https://doi.org/10.14778/3594512.3594529

[88] Yongjun Zhao, Huaxiong Wang, and Kwok-Yan Lam. 2021. Volume-Hiding Dynamic Searchable Symmetric Encryption with Forward and Backward Privacy. Cryptology ePrint Archive, Paper 2021/786.

[89] Leqian Zheng, Lei Xu, Cong Wang, Sheng Wang, Yuke Hu, Zhan Qin, Feifei Li, and Kui Ren. 2023. SWAT: A System-Wide Approach to Tunable Leakage Mitigation in Encrypted Data Stores. arXiv:2306.16851 [cs.CR]

[90] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *Proc. of USENIX NSDI*.

[91] Mingxun Zhou, Elaine Shi, T-H. Hubert Chan, and Shir Maimon. 2022. A Theory of Composition for Differential Obliviousness. Cryptology ePrint Archive, Paper 2022/1357. https://eprint.iacr.org/2022/1357

[92] Mingxun Zhou, Mengshi Zhao, T-H. Hubert Chan, and Elaine Shi. 2023. Advanced Composition Theorems for Differential Obliviousness. Cryptology ePrint Archive, Paper 2023/842. https://eprint.iacr.org/2023/842