Waffle: An Online Oblivious Datastore for Protecting Data Access Patterns

Sujaya Maiyya
University of Waterloo

Sharath Vemula
UC Santa Barbara

Divyakant Agrawal
UC Santa Barbara

Amr El Abbadi
UC Santa Barbara

Florian Kerschbaum
University of Waterloo

ABSTRACT

We present Waffle, a datastore that protects an application's data access patterns from a passive persistent adversary. Waffle achieves this without prior knowledge of the input data access distribution, making it the first of its kind to *adaptively* handle input sequences under a passive persistent adversary. Waffle maintains a constant bandwidth and client-side storage overhead, which can be adjusted to suit the application owner's preferences. This flexibility allows the owner to fine-tune system parameters and strike a balance between security and performance. Our evaluation, utilizing the Yahoo! Cloud Serving Benchmark (YCSB) benchmark and Redis as the backend storage, demonstrates promising results. The insecure baseline outperforms Waffle by a mere 5-6x, whereas Waffle outperforms Pancake—a state-of-the-art oblivious datastore under passive persistent adversaries—by 45-57%, and a concurrent ORAM system, TaoStore, by 102x.

1 INTRODUCTION

More than 94% of current enterprises – including 83% of health care organizations [21] – rely on cloud services, especially for their infrastructure and data storage needs [9]. Organizations outsource their storage to third party cloud providers because of the high cost associated with owning and maintaining an on-premise storage or compute fleet. However, outsourcing an application's data in plaintext can reveal sensitive information to a potentially non-trustworthy cloud provider. While encrypting the data forms the first obvious solution to ensure data privacy, a growing body of attacks [7, 14, 18–20, 23–26, 37, 38, 44, 56] exploit access patterns (e.g., access frequency or duration) on encrypted data to uncover plaintext data. Such attacks are called *access pattern attacks*.

Oblivious RAM [16], or ORAM, is a well known cryptographic technique that mitigates access pattern attacks. While a large number of datastores have integrated ORAM to guarantee obliviousness (e.g., [8, 11, 34, 47, 48, 50, 51]), all these systems incur a lower bound bandwidth overhead of $logN$ [16, 31, 41]. The lower bound implies that for each client request accessing a single object, the ORAM schemes require $\Omega(logN)$ bandwidth overhead, where N is the database size. This overhead can easily throttle system performance when the database size grows to billions of records or when clients send thousands of concurrent requests. These lower bounds exist because ORAM assumes an *active* (but non-malicious) adversary who can not only observe accesses to encrypted data, but also inject queries. Although this adversarial model captures sophisticated adversaries, it hinders performance.

Work to appear in SIGMOD 2024.

Pancake [17] introduced the notion of a *passive persistent adversary* who can persistently view ongoing accesses on encrypted data but cannot inject queries; Pancake ensures obliviousness under this threat model. By assuming a less stringent yet realistic adversarial model, Pancake achieves 229x higher performance than PathORAM [51], a well known ORAM scheme.

However, Pancake has several limitations. First, Pancake achieves, what we term, ***offline obliviousness***: the scheme requires (near) accurate prior knowledge of the original access distribution based on which it creates *uniform* frequency distribution over all outsourced data objects. Any changes in the distribution must be detected immediately to guarantee security. Second, Pancake requires an $O(N)$ cache at the client side, i.e, at a trusted proxy, where for certain distributions, the cache may store close-to N objects locally (§2 provides details). Third, Pancake's security model assumes that client requests are drawn *independently* from a known distribution; it fails to capture scenarios when clients query from unknown distributions or send correlated queries. Because of this, Pancake can only handle *single maps* where each object has a single value (contrary to relational data where each object can have multiple attribute values). Fourth, Pancake's obliviousness algorithm cannot be easily tweaked to weaken the security in a controlled fashion in exchange for potentially better performance.

In terms of obliviousness, Pancake provides obliviousness of access *frequencies*. On the contrary, ORAM provides obliviousness of access *sequences*. This means that given specific sequences, i.e., queries drawn independently from a known distribution, Pancake ensures that the access frequency of each outsourced object is $1/N'$, where N' is the outsourced database size. In fact, for small N's, Pancake's security can be compromised when the input sequence has correlated queries [38]. Meanwhile, ORAM takes *any* sequence of accesses and generates server accesses distributed *uniformly* over the outsourced objects.

Our goal is to design an algorithm that ensures obliviousness of access *sequences*, in-line with ORAM, under a passive persistent adversarial model. But providing *completely* uniform accesses on the server is expensive, as proven in the ORAM's $logN$ lower bounds [16, 31, 41]. Therefore, we weaken this *complete* uniformity to *'somewhat'* uniformity and provide a concrete definition and bounds on the term *'somewhat'*.

Overall, our goal is to design an algorithm and build a system that (i) provides ***online obliviousness*** of access sequences by adapting to any input sequence, (ii) allows tuning security in exchange for performance, (iii) uses a bounded cache, and (iv) achieves the above properties with constant bandwidth overhead.

Our contributions: We make three contributions in this work:

(1) We define a new security model by introducing a new data access uniformity definition called α, β-*uniform*. At a high level, this definition captures how uniform are the accesses to the server. As one of our main design rationales is to provide more flexibility to the application owner and allow them to tune system parameters to trade-off security for performance, the new security definition captures these system parameters such that the extent of the obliviousness guarantees depend on the values set for these parameters.
(2) We design and build Waffle, an oblivious datastore that guarantees α, β uniformity and provides ***online obliviousness*** for queries drawn from *any adversarial chosen sequence* of requests. Waffle is the first system to make no assumptions about input access distributions under passive persistent adversaries. Waffle requires a constant-size cache and has a constant bandwidth overhead, both of which can be configured by the application owner. Moreover, Waffle can be easily extended to handle multi-maps wherein each object can have multiple associated values (e.g., relational data).
(3) We extensively analyze Waffle by conducting a variety of experiments, which indicate that with Waffle's system parameters mirroring that of Pancake, Waffle performs **45.5-57.7**% better than Pancake and **102x** better than TaoStore, while the insecure baseline performs **5.8x** and **6.04x** better than Waffle. We also analyze Waffle's security vs. performance trade-offs and show that Waffle protects against correlated query attack [38] that Pancake is vulnerable to.

2 BACKGROUND

This section provides a background on access pattern attacks and a high level description of Pancake [17] and PathORAM [51].

Terminology: We first explain the meaning of the commonly used terminologies in the paper. *Query* is a get/put request issued by clients. The terms *query* and *request* are used interchangeably. *Plaintext query distribution* or *real distribution* corresponds to the distribution from which the client queries are drawn. *Ciphertext query distribution* corresponds to the access distribution observable by an adversary on the encrypted objects stored at the server. *Popular (or unpopular) objects* are plaintext objects that have a high (or low) probability of being requested by clients. We interchangeably use *object* k_i or *key* k_i to refer to an object whose identifier is k_i.

Access pattern attacks: A growing number of attacks from the security and privacy community [3, 7, 14, 18–20, 23–27, 32, 36–38, 44, 56] have shown that the access pattern of outsourced data, even after encryption, can help an adversary uncover the underlying plaintext data or even the plaintext queries issued by the application. These attacks apply techniques such as *frequency analysis* and l_p-*optimization* [27, 32, 36] on deterministically encrypted databases such as CryptDB [45] to uncover the plaintext data. Note that most encrypted databases encrypt the data object identifiers (e.g., primary keys) using deterministic encryption so as to retrieve the appropriate object from the external server.

The *frequency analysis* attack, a fundamental inference attack technique, relies on the ciphertext space C and the message space $\mathcal{M}$. The attack operates by correlating or assigning the most frequently occurring elements in a deterministically encrypted column c over C with the most frequently occurring elements in an auxiliary dataset m over $\mathcal{M}$.

Pancake: *obliviousness of access frequencies.* Pancake [17] mitigates access pattern attacks by employing *frequency smoothing* to flatten and make the ciphertext access frequency distribution uniform. Pancake requires (i) prior knowledge of the plaintext access distribution and (ii) the queries to be drawn independently, to ensure obliviousness. It employs two main techniques: 1). Replicate popular objects such that the system accesses different replicas each time a client queries popular objects; and 2). Add fake queries to real queries to increase the access probabilities of unpopular objects.

Although Pancake presents a mechanism to handle changing distributions, the new distribution must be learnt before ensuring frequency smoothing. We call this ***offline obliviousness*** because Pancake's two strategies for frequency smoothing can only be applied for a known input distribution. Moreover, because it replicates popular objects, propagating updates to all replicas of an object requires Pancake to maintain a datastructure, updateCache, that can grow to size N, wherein in the worst-case *all data objects and their values must be stored locally* at the proxy.

Because Pancake only hides access frequencies, it uses **static key assignment**, i.e., a plaintext key k (or object identifier) always maps to the same storage identifier e_k used to query the object from the server. For small datasets, static key assignments are vulnerable to attacks when the input distribution consists of correlated queries [38] (e.g., an adversary that picks correlated queries to $k1$ and $k2$ in the input distribution observes that e_{k1} and e_{k2} are accessed together with a high probability). However, we note that for settings that hold Pancake's assumption of queries drawn independently from a known distribution, it effectively hides user access patterns.

ORAM: *obliviousness of access sequences.* Because Waffle's adversarial model differs from ORAM's (in which an adversary can inject queries and note the resulting access pattern), we provide a high level discussion of PathORAM [51], a popular ORAM scheme, on only those details that are relevant to Waffle. At inception, PathORAM initializes the external storage in a binary tree by assigning objects to randomly chosen paths. PathORAM maintains a positionMap that stores object ids to path ids on which the object resides – *this is the plaintext identifier to storage identifier in PathORAM.*

Each request to an object results in PathORAM fetching the entire path where the object resides. After each access to an object i, PathORAM deletes i from its previous path and writes it to another randomly chosen path and updates the mapping in the positionMap. This way, each time i is accessed, the adversary observes access to a random path, thus breaking correlations between plaintext object ids to their storage ids. We call this a ***non-static* assignment of plaintext object identifiers to storage identifiers** because after each access to an object, its storage identifier, which is a path id, changes. This non-static assignment helps ORAM hide the server access patterns for ***any input sequence***, including correlated queries. However, this incurs $logN$ bandwidth overhead [16, 31, 41]. For example, PathORAM reads and writes an entire path of a binary tree consisting of $logN$ nodes to serve each client request.

3 SYSTEM AND THREAT MODEL

3.1 System Model

We present Waffle as a key-value store that supports single object get, put, or delete operations. However, Waffle's design can be easily applied to other types of database that support similar single-object operations. Waffle stores the data in an external untrusted storage server and executes requests from clients by routing the queries through a trusted proxy. The use of a trusted proxy is a commonly employed technique [11, 17, 34, 45, 48, 50, 51]. Waffle's design and security guarantees also hold for a system with a single client and no proxy. However, we use the proxy model to support multiple clients requesting data concurrently. The proxy is a stateful entity assumed to be highly available (which can be ensured with techniques such as a primary-secondary replication [15] or a quorum replication [29]; however, this choice is orthogonal to Waffle's goals).

The clients and the proxy reside within the same trusted administrative domain. This is a reasonable assumption as the application, which may not possess enough resources to host and manage all of its data, may host a proxy with significantly lower storage space compared to the server.

Each plaintext data object consists of a key k and a value v. Because Waffle can handle correlated queries, it can be easily extended to support multimaps wherein each key has multiple associated values (we discuss correlated queries in §8.3.2). The proxy encodes a key k using *pseudo random functions* (PRFs) $prf(k, a_k)$, where a_k is auxiliary information of k. $prf(k, a_k)$ acts as the storage identifier for k. The proxy encrypts the values using authenticated encryption $E(v)$. PRFs are deterministic functions, i.e., invoking a PRF with the same input any number of times will result in the same output. However, invoking the same PRF with different inputs generates random output strings. The proxy stores the secret-keys used for $prf(k, a_k)$ and $E(v)$. To avoid attacks based on the length of the object values, we consider all values to be of equal length.

3.2 Threat Model

Waffle assumes the adversary $\mathcal{A}$ to be semi-honest (or honest-but-curious) but non-malicious who can observe and/or record request patterns. $\mathcal{A}$ can use any prior knowledge and the observed access patterns to launch inference attacks. We further assume that the communication layer is unreliable, asynchronous, and insecure. The adversary can view or delay, but eventually delivers, encrypted messages exchanged between the trusted and untrusted domains. The communication layer uses TLS [53] to prevent data tampering or eavesdropping from an attacker who intercepts communication channels. Waffle assumes that the adversary cannot inject queries or manipulate data in transit or storage.

Non-goals: Waffle tolerates no malicious adversaries (such as in blockchain settings) nor 'active' adversaries (as in ORAM) who can inject queries. Waffle also does not aim to protect an application from timing based side channel attacks or implementation based backdoor attacks.

4 WAFFLE OVERVIEW

In this section, we provide a rationale for Waffle's design choices and an overview of the system. The four goals of Waffle are to: i) ensure obliviousness under a passive persistent adversarial model for any sequence of accesses, without requiring prior knowledge of the input distribution; ii) enable the application owner to have more control over the datastore and allow trading security for performance; iii) use a bounded cache, and (iv) have a constant bandwidth overhead.

At a high level, Waffle achieves obliviousness by padding real requests with fake requests. We first consider a simple strawman design where the proxy is stateless (except to store the encryption-keys) and stores no data objects or meta-data. All objects are encrypted before being stored at the server, and every client request accesses the requested object from the server, along with the objects padded for fake queries. The naive solution assumes a static assignment of plaintext object ids to storage ids (similar to Pancake). By adding '*enough*' fake queries per real query and by picking the '*right*' objects for fake queries, this naive solution can ensure obliviousness. We next discuss the challenges of achieving obliviousness in the strawman that led to the design choices of Waffle.

Challenge 1: Batching real queries with fake queries. The naive solution stores all data objects in the server and each client request has to access the data from the server. Accessing a single data object from the server in each server access allows an adversary to distinguish between real vs. fake queries (e.g., the first request made to the server after a period of rest likely corresponds to a real request). Hence, we need to batch real and fake queries together and perform batched server accesses. Deciding a secure yet efficient batch size forms the first challenge of the naive solution.

Solution: Given our goal of providing the application owner with more control over the data system, Waffle allows the application to choose the batch size, B (> 1). Because the proxy and the server reside in different trust domains, higher system performance favors reducing proxy to server communications. Therefore, Waffle waits to receive R (≥ 1) client requests before creating a batch of B requests to be sent to the server. However, accessing the same object multiple times in the same batch leaks information on data contention. Waffle hides this by deduplicating and only adding requests to unique objects in R, whose size is represented by r, to the batch. Waffle further allows an application to initialize the database with D (≥ 0) dummy data objects (we explain the advantage of having dummy objects in Challenge 3). Waffle then appends f_D (≥ 0) fake queries on *dummy* objects and $f_R = B - (r + f_D)$ fake queries on *real* objects to the batch. Essentially, each batch sent to the server consists of r real queries on real objects, f_R fake queries on real objects, and f_D fake queries on dummy objects, and the parameter values of B, R, and f_D are all chosen by the application. Having discussed how Waffle constructs a batch, we next discuss what objects to choose for fake queries in a batch.

Challenge 2: Choosing objects for fake requests. Apart from replicating popular objects (§2), Pancake [17] primarily relies on fake queries to ensure obliviousness. Because Pancake knows the real query distribution, it inverts the distribution and chooses objects for fake queries from this inverted distribution. However, since Waffle assumes nothing about real query distribution, deciding what objects to choose for fake queries is challenging because picking popular objects for fake queries can aggravate the access probability imbalance between popular and unpopular objects.

Solution: Waffle adapts a dynamic approach of continually maintaining how recently a given object was accessed using (integer) timestamps. It chooses least recently accessed objects, i.e., objects with the least timestamps, for fake queries while creating a batch. Note that maintaining the timestamp information makes the proxy stateful. Each access to an object on the server, either as a real or a fake query, updates its access timestamp. Waffle uses a balanced binary search tree (BST) to maintain access timestamps because it efficiently supports `search` and `update` operations: an object with minimum timestamp object can be found in constant time and its timestamp can be updated in $logN$ time. While tracking access timestamps in a binary search tree helps pick least recently accessed objects for fake queries, ensuring obliviousness for highly skewed accesses remains an open challenge.

Challenge 3: Handling highly skewed accesses. Real-world accesses typically exhibit highly skewed access patterns [1, 10, 42] where most user requests access a small subset of data. Because the naive solution stores all the data objects only at the server, ensuring obliviousness will require adding a prohibitively large number of fake queries. For example, consider an extreme case where all user requests are to a single data object, o_i, out of a million objects. Ensuring any sense of uniform access will require adding many fake queries per real request; otherwise (encrypted) o_i will be accessed significantly more often than the rest, rendering this solution insecure.

Solution: Waffle employs client-side caching, i.e., at the proxy, to mitigate the above challenge. Essentially, Waffle caches all real objects ($r + f_R$) accessed in a batch and serves user requests from the local state if the requested object resides in the cache. When popular objects reside in the cache, r reduces and f_R increases, causing more unpopular objects to be accessed per batch, thus improving security (because no object remains un-accessed for long). This implies that larger cache ensures higher security. Reiterating our goals of providing more control to the application, Waffle allows the application to choose the cache size and ensures that the cache size remains bounded, in line with our other desired goal (this is unlike Pancake whose cache size can grow to $O(N)$).

Waffle uses a least-recently-used (LRU) cache to retain frequently accessed objects in the cache (we explain what happens to evicted objects in Challenge 4). To retain popular objects in the cache, Waffle adds dummy objects to the database and access f_D of them in each batch. This reduces the number of real objects accessed in each batch, in-turn reducing the number of cache evictions necessary to cache real objects. Moreover, dummy objects introduce another layer of security since an attacker must first distinguish between real and dummy objects before performing inference attacks.

Challenge 4: Ensuring online obliviousness with bounded cache. Even when popular objects are cached, certain access sequences can violate obliviousness due to the static assignment of plaintext object ids to storage ids. We explain this security challenges with a simple example where the database consists of N objects, the cache size is 3, and the users access objects o_1, o_2, o_3, o_4 in a repeated sequence. In such request patterns where the size of popular objects is slightly larger than the cache size, every client request will result in a cache miss, needing to fetch the object from the server. Hence, the frequency of these 4 (encrypted) objects on server will be higher that the rest. The adversary can perform an attack by choosing similar access sequences and can essentially identify the plaintext object id to storage id mappings. We note that even if we allow the cache size to grow dynamically instead of fixing the size, similar attacks can be performed depending on the cache eviction policy unless we either make assumptions about knowing the real distribution or storing all accessed objects in the cache without ever evicting them.

Solution: We address this challenge by making the plaintext id to storage id assignments *non-static*, similar to ORAM schemes (§2). Essentially, Waffle updates a key k's storage id after each access to k, which means, each object stored at the server is written and then read **at most once** before its storage id changes (the old object can be deleted after reading it to save storage space; however, this choice has no impact on security).

Waffle achieves this as follows: (i) To access each outsourced object, it generates the storage id by invoking the PRF with both the object's plaintext key k and its current access timestamp ts_k maintained in the BST, i.e., $e_k = prf(k, ts_k)$. At initialization, $ts_k = 0$. (ii) After each access to k, Waffle updates its access timestamp to ts'_k in the BST and stores the retrieved object locally in the cache. To save storage space, Waffle *deletes* the object with id e_k from the server. (iii) If and when the cache evicts object k, the proxy writes $e'_k = prf(k, ts'_k)$ along with k's encrypted value to the server. The server cannot distinguish if e'_k and e_k correspond to the same plaintext key k. This technique implies that an object *either only* resides in the cache or at the server. To push the evicted objects back to the server, Waffle always reads B objects and writes B objects (but not necessarily the same objects). This technique of reading and writing objects accessed by clients is a commonly used method [11, 17, 34, 48, 50, 51].

Summary: To summarize, Waffle (i) relies on non-static assignment of plaintext ids to storage ids and accesses each storage id, and hence each encrypted object, at most once for security, (ii) accesses objects in batches consisting of real and fake queries on real and dummy objects, (iii) picks objects for fake queries based on the recency of access, and (iv) caches popular objects to reduce the number of fake queries necessary to ensure obliviousness. Having provided an overview of Waffle's design choices, we next present the formal security definition of Waffle.

5 SECURITY MODEL

Waffle employs the same adversarial model defined in Pancake [17]: *passive persistent adversary*. The adversary can observe all accesses to the outsourced encrypted data, but it cannot inject its own queries. The adversary can record the access patterns over time and use it to perform inference attacks. The adversary can choose the plaintext query distribution or sequence of accesses from which client queries are drawn from, and it can arbitrarily change the distribution (however, the adversary cannot compromise clients to realize individual queries picked from the input distribution).

The persistent passive adversarial model is weaker than that of ORAM's [16], in which adversaries can also inject queries. On the other hand, this model is stronger than the *snapshot* security model where the adversaries can only access snapshots of the database without persistently observing query accesses [28, 39, 43].

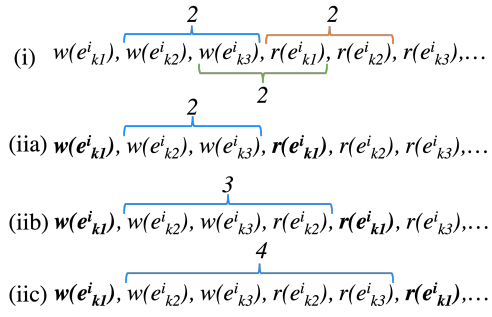


Figure 1: r and w indicate reads and writes; values are omitted for brevity. (i) depicts a *fully* uniform access sequence whereas ii(a)-ii(c) depict a *somewhat* uniform access sequence.

Waffle aims to hide access sequences and not just access frequencies (see discussion in §2) by employing a non-static assignment of plaintext ids to storage ids, i.e., Waffle updates an object's storage id after each access to the object. Essentially, each object stored at the server is written and then read **at most once** before its storage id changes (to minimize storage, Waffle deletes the old id and its object). This implies a smooth access *frequency* of one write and one read per object on the server. However, an adversary can still observe the number of accesses *between* the write and the subsequent read of an object. This can lead to attacks wherein an adversary can map the encrypted objects written at initialization but never accessed to the plaintext objects not accessed in the (adversary chosen) input sequence.

To be oblivious under a passive persistent adversary, a data system must ***bound the number of server accesses*** within which a written object must be read. This generates a *somewhat* uniform access since every outsourced object is accessed in *at most* a bounded number of accesses, α, after it was written. On the contrary, a *completely* uniform access would guarantee that every outsourced object is accessed *exactly* at some α' accesses after it was written.

Figure 1 depicts the difference between complete vs. somewhat uniform server accesses. Consider a database with 3 objects, with plaintext keys $k_1 - k_3$. $e^i_{k_j}$ represents the storage id generated using $prf_i(k_j) = prf(k_j, ts_i)$, i.e., the i^{th} access of k_j. Figure 1(i) is *completely* uniform because each written object is read after exactly $\alpha' = 2$ accesses. This ensures complete obliviousness because each outsourced object is accessed exactly at the α'^{th} access from when it was written, generating a uniform server access pattern regardless of the client request pattern. Whereas, Figures 1 (iia) to (iic) depict *somewhat* uniformity, wherein a written object is accessed in *at most* $\alpha = 4$ accesses. This opens up different combinations of accesses, especially for large α values, yet ensures (a weaker notion of) obliviousness because every written object will be accessed within α accesses. However, larger αs leak more information on the timing of accesses visible to an adversary, and is less secure. We next formally define this notion of *somewhat* uniform.

5.1 α, β uniformity definition

Let read(k, v) and write(k, v) be accesses. For brevity we drop v in our notation for security analysis, since an encrypted v is a random bit string, and use $\tau(k)$ to denote either a read or write access, where $\tau \in \{\text{read}, \text{write}\}$. Let S_{Proxy} be the sequence of accesses $\tau_1(k_1), \tau_2(k_2), \ldots$ received by the proxy from clients. Let S_{Server} be the sequence of accesses $\tau'_1(prf_1(k_1)), \tau'_2(prf_2(k_2)), \ldots$ received by the server from the proxy, where prf_i denotes the i^{th} invocation of the PRF (i.e., $prf_i(k) = prf(k||ts_i)$ where ts_i is k's timestamp after i accesses). The server sequence generated by the proxy can (i) omit accesses from the client if objects reside locally in the proxy; and (ii) insert accesses not requested by clients (e.g., fake queries).

The proposed uniformity-based security definition called α, β-*uniformity* bounds the maximum number of server accesses, α, after which a written object will be read, and the minimum number of server accesses, β, after which a read object can be written. We call any sequence of accesses that satisfies the α and β bounds α, β-*uniform*. We present the formal definition below.

Definition 1. *Let S_{Server} be the (infinite) sequence of accesses $\tau'_1(prf_1(k_1)), \tau'_2(prf_2(k_2)), \ldots$ received by the server from the proxy. We say S_{Server} is α, β-uniform if*

(1) *For any $e^i_k = prf_i(k) : \tau'_i = write(e^i_k)$ in S_{Server}, there exists a j^{th} access $\tau'_j = read(e^i_k)$ in S_{Server} with $i < j \leq i + \alpha + 1$. No τ'_k that reads or writes k exists with $i < k < j$.*
(2) *For any $\tau'_i = read(prf_i(k)), \tau'_j = write(prf_j(k))$, with $i < j$ in S_{Server}, $i + \beta < j$ holds. No τ'_k that reads or writes k exists with $i < k < j$.*

*S_{Server} is α, β-uniform if the bounds hold for **all** keys $k \in N$.*

This definition implies a two-way bound: in the sequence of accesses S_{Server} received by the server (1) if the i^{th} access wrote a key $prf_i(k)$, then between $i + 1$ and $i + \alpha + 1$, a j^{th} access will read that key; and (2) if the i^{th} access read $prf_i(k)$, then the next write request, $prf_j(k)$ (where $j > i$) occurs at least after β accesses from i. In an α, β-uniform sequence, there exists a write between two subsequent reads of an object and vice versa. The values of α and β correspond to the overall number of proxy to server accesses. If the proxy accesses objects in batches, α, β, i, and j correspond to the respective batched server accesses (and not individual object accesses).

Note that in the definition, α is an upper bound and β a lower bound (whose values for Waffle are discussed in §7). The lower bound for α is 0 because an object written in one round can be accessed in the next round. We chose not to provide an upper bound for β because a proxy design may cache popular objects indefinitely in the cache. Also, Waffle maintains that each key e^i_k at the server is written and read exactly once; deleting e^i_k from the server after reading it has no security implications. The choice of the parameters α and β allows the administrator to tune the trade-off between security and privacy. We next discuss why an α, β-uniform sequence ensures obliviousness using a theorem.

Theorem 5.1. *If a sequence of accesses, S_{Server}, generated by a proxy is α, β-uniform with $\alpha = 0$ and $\beta = 2(N - 1)$, then S_{Server} hides the client access pattern, ensuring complete obliviousness.*

Proof. Since α dictates the maximum number of accesses after which a written object **must** be read, minimizing α enhances obliviousness (as even unpopular objects will be read sooner for lower values of α). On the flip side, we want to maximize the β value,

which dictates the minimal interval guaranteed between reading and writing back an object. Maximizing β reduces the frequency with which an object is accessed on the server (i.e., if we delay writing back an object, we can delay reading it). The minimum value of α is 0; whereas the maximum value of β is $2(N-1)$ because the maximum a proxy can delay writing back a read object, k_1, is after reading and then writing the other $N-1$ objects. This generates an access sequence such as (with α and β marked for k_1 and keys represented as plaintext for readability):

$$\langle \underbrace{w(k_1), \mathbf{r(k_1)}}_{\alpha=0}, \underbrace{w(k_2), r(k_2), w(k_3), r(k_3), \cdots, w(k_n), r(k_n)}_{\beta=2(N-1)}, \mathbf{w(k_1)} \rangle$$

Note that k_1 to k_n can be any permutation of the N keys and that this theorem assumes no batching, hence each server access either writes or reads one object.

For S_{Server} to be α, β-uniform with $\alpha = 0$ and $\beta = 2(N-1)$, the sequence **must** continue this pattern of writing and then reading the N objects in the same order, *irrespective of what objects the clients request*. Not immediately reading after writing k_i will violate the $\alpha = 0$ bound; whereas, writing k_i within $2(N-1)$ accesses after reading it will violate the $\beta = 2(N-1)$ bound. Such a sequence ensures complete obliviousness because (i) it is independent of the client request pattern, and (ii) the adversary controlling the server observes a deterministic data access pattern where an encrypted object is read *exactly* in the next access after it was written (similar to Figure 1(i)). □

While setting $\alpha = 0$ and $\beta = 2(N-1)$ provides complete obliviousness, this is impractical for two reasons: (i) client requests cannot be served within a reasonable time because some requests may have to wait for $2N$ server accesses; (ii) the proxy temporarily needs to store all N read objects before it can write them back. Waffle aims to maximize security while still being a practical data system by generating *somewhat* uniform accesses. Specifically, Waffle allows an administrator to choose system parameters that can enhance security at the cost of performance/proxy storage and vice versa. We give a detailed analysis of α and β in §7 and prove that Waffle is α, β-uniform, both theoretically §7 and experimentally §8.3.

6 WAFFLE

This section explains Waffle in detail. Table 1 lists all the variables used in explaining the protocol. All variables in Table 1 except f_R are fixed system parameters set by the application owner.

6.1 Initialization

At inception, Waffle receives a set of N key-value pairs, $\langle keys, values \rangle$, from the application and initializes both the stateful data structures at the proxy and the data itself at the server as follows.

Binary search trees *(proxy)*: The proxy first generates D random dummy keys (that are unique) and values of the same length as that of real objects. The proxy initializes two balanced binary search trees (BSTs): one for N real objects and one for D dummy objects. The BSTs are initialized by setting the access timestamps, ts, of all keys to 0. The trees are balanced on $\langle ts : plaintext\_key \rangle$. Note that although the BSTs are $O(N)$ data structures, storing them on the proxy incurs low storage overhead since they only store constant-size timestamps and not object values (e.g., storing timestamps for 1M objects requires 8MB of storage at the proxy.

Symbol	Meaning
D	Number of dummy objects in the system
B	Batch size of requests sent to the server
R	Maximum no. of real queries on real objects in each batch
f_R	Number of fake queries to real objects in each batch
f_D	Number of fake queries for dummy objects in each batch
C	Cache size

Table 1: Variables used in Waffle.

Cache *(proxy)*: Next, the proxy randomly chooses a set of C (the preset cache size) key-value pairs from $\langle keys, values \rangle$ and initializes the cache with these objects. Although the cache size can be 0, a meaningful minimum cache size is $R + f_R$ to allow storing all the real objects accessed in a batch.

Database *(server)*: The proxy merges the remaining $N - C$ objects and the D dummy objects and shuffles them. It then encodes each key-value pair, $\langle k, v \rangle$, as: $\langle prf(k, ts_k), E(v) \rangle$, where ts_k is k's access timestamps (=0) and E is any authenticated symmetric-key encryption scheme. The proxy then initializes the database in the server with the encoded key-value pairs. Waffle incurs a storage overhead of D compared to its plaintext counterparts.

6.2 Protocol

After the initialization process, Waffle can start serving client requests. Algorithm 1 explains how Waffle serves client requests. At a high level, upon receiving R unique requests but not necessarily to unique objects, the algorithm executes a batched read (lines 2-29) followed by a batched write (lines 30-44). Processing each batch of R requests increments Waffle's global timestamp, ts (line 5). This is used to identify the least recently accessed objects for fake queries. The algorithm relies on two temporary data structures, *cliResp* and *dedupReqs* to serve client queries. *cliResp* helps map client request ids (which are unique) to their responses. For security, each batch must access unique objects; otherwise the number of repeated accesses to an object can leak information. *dedupReqs* stores the deduplicated requests for each unique object in the current batch to ensure a response is sent to these requests.

Read phase: This phase creates a batch of unique (encoded) keys to be read from the server. For each of the R requests, if the cache contains the requested object, then the algorithm serves read requests from the cache and updates the cached value for write requests. If a request results in a cache miss, the algorithm adds this request to *dedupReqs* to track any future duplicates. However, for each cache-missed read request, the algorithm notes that this request needs a response from the server (line 10), whereas for write requests, the algorithm marks the *need_resp* flag as false (line 13) because write requests do not need a response from the server. Note that if a written object is not in the cache, the algorithm adds it to the cache (line 13).

Adding a written object to the cache even before the object is fetched from the server is important to ensure **linearizability** [22] – a database consistency guarantee that implies operations on an object appear to take place instantaneously and all operations appear *linear*. This means a read of an object after a write on that

object must reflect the updated value, even when the two requests are within the same batch. Waffle ensures this by caching an uncached written object and reading the cached value for subsequent read requests in that batch (line 7). Storing written objects in cache also implies that *while a batch is being processed, an object may reside both at the proxy and the server*. In such cases, the cache always stores the latest value of an object. This also implies that temporarily the cache size can grow to at most $C + R$ objects.

After processing R client requests, the algorithm identifies the $r(\leq R)$ unique plaintext keys whose values must be fetched from the server and encodes them by invoking the pseudo-random function with the plaintext key k and its current access timestamp, ts_k, i.e., $prf(k, ts_k)$. This is the index or the identifier based on which Waffle accesses objects on the server. The algorithm tracks these encoded keys in *readBatch* (line 18).

The algorithm then adds two sets of fake queries to the batch: one set for dummy objects and another for real objects. In particular, it adds a fixed number, f_D, of fake queries to dummy objects and a variable number, f_R of fake queries to real objects. f_R depends on r, the number of unique real objects that need to be fetched from the server. Since our batch size is fixed to B, $f_R = B - (r + f_D)$. Note that, as mentioned in Challenge 2, the algorithm uses the BSTs to pick real or dummy objects with least access timestamps for fake queries (lines 21 and 26) and each time it accesses an object from the server as part of a real or a fake query, the algorithm updates the access timestamp of that object to the current timestamp (lines 19, 23, and 28). For dummy objects, however, the algorithm sets the timestamps of *all* D objects to the latest timestamp ts after every $\frac{D}{f_D}$ batches, i.e, each time all D objects are accessed. This randomizes the order in which dummy objects are picked for the next set of $\frac{D}{f_D}$ batches. After appending encoded keys for fake queries to *readBatch*, the algorithm reads a batch of B objects from the server. As explained in Challenge 4, for security, Waffle can access each outsourced object at most once before updating its identifier (achieved via invoking the PRF with updated access timestamps). Because of this, a background thread deletes the B accessed objects from the server after receiving the read response. Note that deleting these objects has no security implications; Waffle deletes these objects to bound the database size to $O(N)$.

Write phase: This phase processes all the received responses from the server and creates a batch of encoded-key to encrypted-value pairs to be written on the server. First, the algorithm generates responses to the deduplicated requests whose object values were retrieved from the server (lines 33-36). Next, the algorithm proceeds to cache all the $r + f_R$ real objects accessed in the batch (lines 37-41). Because Waffle employs a bounded cache, the algorithm first evicts an object from the cache before adding a new object. All evicted objects must be written back to the server but with new storage ids (or encoded keys). Waffle guarantees this because whenever the objects being evicted were read, the read phase would have updated their timestamps. Therefore, invoking `GetIndex` in line 39 produces new encoded keys, which are written back to the server along with their (re-)encrypted values using *writeBatch*. *writeBatch* also includes the same dummy objects accessed in *readBatch* but with re-encrypted dummy values (line 43).

Algorithm 1 The algorithm employed in Waffle to obliviously serve client requests. R, B, f_D are system parameters of type integer. Assumption: Cache size is at least $B - f_D + R$ and is initialized with $B - f_D$ random objects.

```
1:  procedure HandleRequests()
2:      upon receiving R client requests
3:      cliResp ← {} ▷ A map of request ids and their responses
4:      dedupReqs ← {} ▷ A map of keys and lists of requests
5:      ts ← ts + 1
6:      for (rId, op, k, val) in R do
7:          if op = read and k in cache then
8:              cliResp[rId] ← cache[k]
9:          else
10:             dedupReqs[k] ∪← (rId, need_resp = true)
11:         if op = write then
12:             if k not in cache then
13:                 dedupReqs[k] ∪← (rId, need_resp = false)
14:             cache[k] ← val
15:             cliResp[rId] ← cache[k]
16:     readBatch ← {} ▷ A map of encoded keys and plaintext keys
17:     for k in dedupReqs.keys() do
18:         readBatch[getIndex(k)] ← k
19:         BST.setTimestamp(k, ts)
    ▷ Add fake queries to dummy and real objects
20:     for i = 1 to f_D do
21:         k ← BST.getMinTimestampObj(dummy)
22:         readBatch[getIndex(k)] ← k
23:         BST.setTimestamp(k, ts) ▷ Set all timestamps to ts
                                      after picking all D objects
24:     r ← dedupKeys.size()
25:     for i = 1 to B − (r + f_D) do
26:         k ← BST.getMinTimestampObj(real) ▷ Ensure k is
                                               not in cache
27:         readBatch[getIndex(k)] ← k
28:         BST.setTimestamp(k, ts)
29:     send readBatch.keys() to server
    ▷ upon receiving server's response
30:     writeBatch ← {}
31:     for (idx, val) in resp do
32:         k ← readBatch[idx]
33:         if k in dedupReqs then
34:             for each (rId, need_resp) in dedupReqs[k] do
35:                 if need_resp = true then
36:                     cliResp[rId] ← val
37:         if objectIsReal(k) then
38:             k′, v′ ← cache.evict()
39:             writeBatch ∪← (getIndex(k′), v′)
40:             val′ ← cache[k] if k in cache else val
41:             cache[k] ← val′
42:         else
43:             writeBatch ∪← (getIndex(k) , ∅) ▷ Dummy object
44:     Send writeBatch to server

45: procedure getIndex(k)
46:     return prf(k, BST.getTimestamp(k))
```

A meaningful minimum cache size is $R + f_R$ to store all real objects accessed in a batch; for cache sizes smaller than $R + f_R$, the algorithm will re-write the objects fetched from the server after re-encrypting them. Note that the real objects written back in each batch may not, and likely will not, correspond to the real objects accessed in the read phase of that batch.

f_R plays a vital role in ensuring security as this is responsible for accessing objects that are rarely queried in the input sequence. Moreover, Waffle incurs a constant bandwidth overhead of $(f_D + f_R)/R$ requests per real request and all 3 parameters can be tuned by the application owner.

Supporting inserts and deletes: The discussions until now focused only on supporting get/put requests. Depending on the number of dummy objects, D, configured by an application, Waffle can support insert and delete requests by swapping dummy objects for real objects for inserts and vice versa for deletes. However, this changes the α, β bounds of the system (§7).

7 SECURITY OF WAFFLE

This section provides the bounds on α and β, which depend on the system parameters of Algorithm 1. Informally, the α upper bound ensures that *any* object written to the server will be read (and then deleted) after at most α server accesses. The β lower bound guarantees that once an object is read (and then deleted) from the server, it will be written back only after at least β server accesses.

Theorem 7.1. *When the proxy uses Algorithm 1, then*

$$\alpha = \left\lceil \max\left(\frac{N-1}{B-R-f_D}, \frac{D}{f_D}\right)\right\rceil$$

Proof. Intuitively, deriving bounds for α requires computing the worst-case distance from when an object, real or dummy, is written to the server and then read. Algorithm 1 ensures that objects in the cache are not stored on the server once a batch's processing completes. Let write(k) be an access to a real key k on the server (and hence it is no longer in the cache). Then that key is read from the server when a client requests it. If no client requests it (within α access to the server), the proxy will request it in Line 26 of Algorithm 1 after at most $N-1$ other real keys have been accessed and their timestamps have been updated. Then that key is certainly the least recently accessed key with the least timestamp in the BST. Since each batch queries at least $f_R = B - R - f_D$ real objects as fake queries, we can derive an upper bound on α for real keys of $\lceil \frac{(N-1)}{B-R-f_D} \rceil$. Note that α is an upper bound and hence we focus on real objects (likely unpopular) accessed via fake queries; popular real objects residing on the server will be accessed as real queries within α server accesses.

Let write(k) be an access to a dummy key k on the server (which is never stored in the cache). The proxy will request it in Line 21 of Algorithm 1 after **at most** $D-1$ other dummy objects have been accessed. Since at least f_D fake queries to dummy keys occur per batch, we can derive an upper bound α for dummy keys of $\lceil \frac{(D-1)}{f_D} \rceil$. Since we allow $D = f_D = 0$, i.e., no dummy keys, which has no impact on α, we use the looser bound $\lceil \frac{D}{f_D} \rceil$ assuming $\frac{0}{0} = 0$. Note that setting $\frac{N-1}{B-R-f_D} = \frac{D}{f_D}$ produces stronger security.
Theorem 7.1 follows from the combination of the two bounds. □

Theorem 7.2. *When the proxy uses Algorithm 1, then*

$$\beta = \left\lfloor \frac{C}{B - f_D + R} - 1 \right\rfloor$$

Proof. The value of β dictates the minimum distance between reads and the following writes of every object. All objects read but not yet written are stored in the cache. Intuitively, deriving β bounds requires us to calculate the earliest an object will be evicted from the cache. To bound writes after reads, we do not need to care about dummy keys, since they can be arbitrarily changed by the proxy. Algorithm 1, which uses a least recently used (LRU) cache, updates the recency position (or factor) of the elements in the cache at two places: (i) in the read phase (lines 8 and 14), and (ii) in the write phase (line 41). For the read phase, the algorithm updates cache positions of at most R elements, when all R client requested (unique) objects reside in the cache. The write phase updates the cache positions of at most $B - f_D$ elements, corresponding to all the real objects accessed in that batch. Hence, the fastest an object will be replaced under the LRU replacement strategy from the cache is after at most $\frac{C}{B-f_D+R}$ rounds of accesses and there are least $\frac{C}{B-f_D+R} - 1$ rounds between adding that object to the cache and replacing it in the cache. Thus Theorem 7.2 follows. □

Theorem 7.3. *For any sequence S_{Proxy} of accesses received by the proxy from clients, the sequence S_{Server} of accesses received by the server, which is produced by Waffle's proxy is α, β-uniform.*

Proof. This proof is a simple deduction from Theorems 7.1 and 7.2. Waffle employs Algorithm 1 to generate server accesses, S_{Server}, for all client requests, S_{Proxy}. Theorems 7.1 and 7.2 prove the bounds on α and β, respectively, on the server accesses, S_{Server}, generated by Algorithm 1. These bounds are independent of the input sequence. Since Waffle uses the same algorithm, the sequence generated by Waffle (i.e., the proxy) is α, β-uniform. □

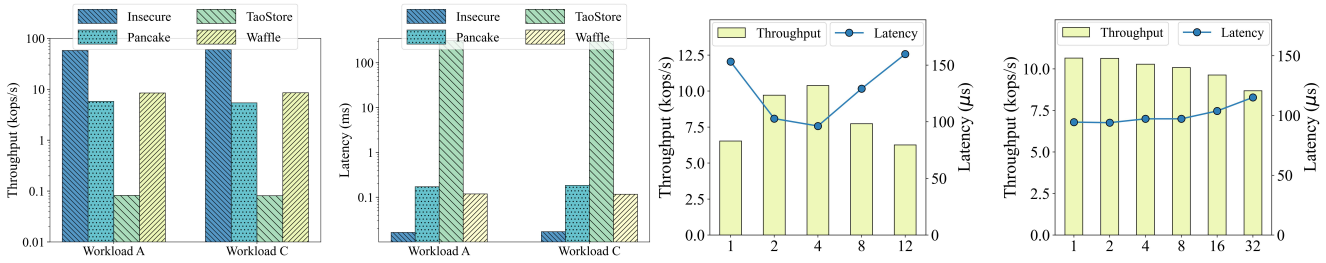
8 EXPERIMENTAL EVALUATIONS

This section studies how Waffle performs both in comparison to other baselines and to different system parameters. In particular, we aim to answer the following questions through experimental evaluations:

(§8.1) How does Waffle perform in comparison to an insecure baseline (to evaluate the cost of privacy) and other oblivious baselines, Pancake [17] and TaoStore [48]?
(§8.2) How does the various systems parameters of Waffle, as defined in Table 1, affect its performance?
(§8.3) What are the security vs. performance trade-offs of Waffle? Can Waffle handle correlated queries?

Experimental Setup: Our setup consists of 3 machines one each for a storage server, a proxy, and a (multi-threaded) client. Each machine has 2x Intel E5-2620v2 CPS with 12 cores, 32 GB RAM, with 10 Gbps Ethernet connectivity. The server deploys Redis [46] as an in-memory key-value store as the backend database. We implemented Waffle in C++. The prototype implementation can be found at: https://anonymous.4open.science/r/waffle_test-8C8C/ReadMe.md

Baselines: We compare Waffle with three baselines:



(a) Waffle vs. baselines (b) Waffle vs. baselines (c) Varying # of proxy cores (d) Varying cache % (of N)

Figure 2: (a) and (b) Throughput and latency of Waffle in comparison to an insecure baseline, Pancake, and TaoStore. (c) When the number of core increase from 1 to 4, Waffle's performance increases. But the multi-threading overwhelms the proxy beyond 4 cores, degrading system performance. (d) Counter-intuitive to expected behavior, Waffle's performance degrades with the increase in cache size: although cache size impacts security positively, the LRU strategy for large caches reduces its performance.

1) An insecure baseline wherein the clients directly store and query data from Redis. This baseline performs no data encryption nor executes any algorithm to ensure obliviousness. This baseline helps identify the performance penalty of achieving privacy.
2) Pancake [17], an oblivious datastore that ensures privacy under a passive persistent adversary. This baseline compares Waffle with the state-of-the-art oblivious datastore under a similar adversarial model as Waffle. We also acknowledge that Waffle's implementation, depending on the component, either builds on top of or reuses Pancake's implementation, which is open-sourced.
3) TaoStore [48], an ORAM-based datastore that can serve concurrent client requests (as compared to the sequential solution of PathORAM). Because Waffle handles concurrent requests, the ORAM baseline must also handle concurrency for a fair comparison. We chose TaoStore as the concurrent-ORAM baseline over other alternatives such as ConcurORAM [8] or Oblivistore [50] because of the implementation availability. We however note that the performances of these schemes are comparable with a difference of ~100 ops/sec. While Snoopy [12] is another recent concurrent-ORAM system, we chose not to use it as a baseline since Snoopy requires trusted hardware, making the comparison with Waffle incompatible. SEAL [14], another potential baseline, is an ORAM scheme with adjustable security; however, Waffle is not compared with it since SEAL only supports single-threaded clients and due to a lack of implementation availability.

Workloads and default parameters: Waffle and the baselines use YCSB [10], a standard benchmarking tool for key-value stores, to evaluate their performances. All experiments use 2^{20} (i.e., 1M) key-value pairs with 8B keys and 1kB values. We compare the performance with 2 YCSB workloads: Workload A consisting of 50% reads and 50% writes and Workload C, with 100% reads, as these two workloads represent the extreme read:write proportions in YCSB. Unless noted otherwise, all our experiments use a Zipf factor of 0.99 mimicking real-world workloads, which exhibit high skewness [1, 10, 42]. Each experiment measures the average throughput (measured for real object) and latency of the system being evaluated.

Waffle's obliviousness algorithm relies heavily on many configurable system parameters (Table 1). We conduct extensive evaluations to measure how these parameters affect system performance, as will be discussed in §8.2. We note the default values used in the experiments here and the reasoning behind these defaults is explained in the following relevant sections: the plaintext database size, $N = 2^{20}$; batch size, $B = 2500$; maximum number of real queries per batch, $R = 1000$ (40% of B); dummy objects, $D = 350k$ (this setting helps maintain high security by keeping the two ratios of α equal); number of fake queries on dummy objects per batch, $f_D = 500$ (20% of B); and the number of proxy cores set to 4.

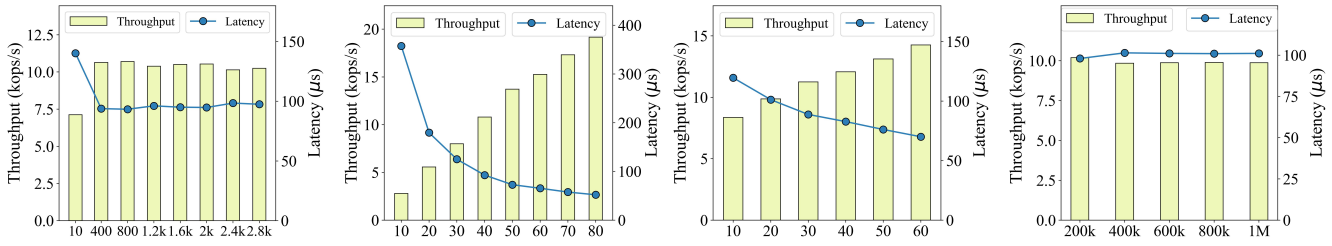
8.1 Comparing with the baselines

This section compares Waffle's performance with that of an insecure baseline, Pancake, and TaoStore; Figures 2a and 2b depict the throughput and latency. This experiment uses a single core proxy for both Waffle and Pancake (we could not run a multi-core proxy in our setup). The reason this experiment use a batch size of 2500 is to be comparable with Pancake's batch size. Although Pancake [17] has an algorithm-level batch size of 3 server requests per client request, their implementation internally batches many of these server requests for added security. We measured the average size of this batch sent to the server, which consisted of 2500 requests, of which 1/2 correspond to real requests (their algorithm tosses a coin and picks a real request with $\delta = 1/2$ probability). Accordingly, for Waffle, this experiment creates batches of (B=) 2500 requests, at most half of which correspond to real client requests, i.e., $R = 1250$, and $f_D = 500$.

The results indicate that the insecure baseline performs between **5.8x** and **6.04x** higher than Waffle, highlighting the cost of achieving obliviousness. Waffle incurs ~3x more latency than the insecure baseline. Meanwhile, for the system configuration that mimics Pancake's, Waffle performs **45.5-57.7**% better in throughput than Pancake, while Pancake requires **45.7-58.6**% higher latency on average to serve a client request. Compared to the concurrent-ORAM baseline, TaoStore, Waffle's throughput is **102x** higher, and while Waffle requires < 1ms to serve client requests, TaoStore requires about 300ms.

8.2 Varying system parameters

This section studies how the different configurable parameters of Waffle affect its performance. Each of the following experiments keeps the default values for all parameters (as mentioned in §8) except vary the parameter in consideration.



(a) Changing batch size B **(b) Changing R (as a % of B).** **(c) Changing f_D (as a % of B).** **(d) Changing D.**

Figure 3: (a) Increasing the batch size has no performance improvement beyond a size of 400 as long as the percent of real and fake queries are consistent across them. (b) Increasing the percent of real requests in a batch significantly improves performance due to accommodating more client requests. (c) Increasing the ratio of fake queries on dummy objects, f_D, improves performance because it reduces the cache insertions and evictions that occur per batch. (d) Changing the number of dummy objects in the system has no impact on performance.

Varying the number of cores: Before evaluating Waffle's behavior when the system parameters vary, we first conduct an experiment by changing the number of cores and measuring its impact on Waffle's performance. As seen in Figure 2c, when the number of cores increases from 1 through 4, the system throughput increases by 58.9% and the latency reduces by 37.2%, reaching the optimal performance for this experiment. Performance declines beyond 4 cores with throughput plummeting by 40%. This indicates that the overhead of multi-threading beyond 4 cores overwhelms the proxy, adversely affecting its performance. Therefore, the following experiments set the number of cores to 4.

Changing cache size: This experiment increases the cache size as a percent of N, the database size, starting with 1% up to 32% (Figure 2d). The performance gradually degrades with the increase in cache size, with the optimal performance, i.e., highest throughput and lowest latency, at a cache size of 1% and 2%. This behavior is counter-intuitive when compared to plaintext datastores, whose performance typically increases with the increase in cache size. The two reasons why Waffle's performance favors smaller cache are: (i) Waffle relies on the cache for security only (β directly depends on the cache size §7) and not to enhance performance. Even when a client requests an object residing in the cache, Algorithm 1 responds to all batched requests at once for security. Hence, algorithmically, cache size has no impact on performance. (ii) Waffle assumes that the deployed cache uses the least recently used (LRU) strategy. The cache needs to track the recency of all objects it stores. The larger the cache size, the more information it tracks, reducing the system performance. Because the 2% cache size is the largest cache size with optimal performance, we set the default cache size to 2% in the experimental setup.

Changing B: This experiment increases the size of the batch, B, sent to the server while keeping R, the number of real requests, at 40% of B and f_D, the number of fake requests to dummy objects, at 20% of B. As seen in Figure 3a, Waffle's performance remains nearly unchanged (a 5% maximum difference) for batch sizes greater than 10 (which has the least performance). This experiment indicates that, while batch size has security implications, it does not impact performance. Since Waffle's performance remains unaffected by batch size, we keep 2500 as its default size, which mimics Pancake's experimental batch size.

Changing R: This experiment increases R, the number of real requests in a batch, as a percentage of $B(= 2500)$ starting from 10% to 80%, while 20% is reserved for f_D. The results in Figure 3b indicate that Waffle's performance improves by **5.8x** when R changes from 10% to 80% of the batch size. This is expected behavior because a larger R incorporates more client requests and reduces the number of fake requests to real objects, thus improving performance. However, Waffle's security favors lower R values (as will be discussed in §8.3). Because of this trade-off, we set R at 40% of B as default.

Changing f_D: This experiment changes f_D, the number of fake requests for dummy objects, as a percent of the batch ($B = 2500$) by increasing f_D from 10% to 60% (the rest 40% is reserved for R). As seen in Figure 3c, Waffle's performance improves with the increase in the number of dummy objects requested per batch. This improvement occurs because the larger the number of dummy objects accessed per batch, the fewer the real objects in the batch, which in-turn reduces the number of cache insertions and evictions, improving the performance. We choose $f_D = 500$ (20% of B) as the default to strike a balance between security and performance, since the security parameter α favors lower f_D values.

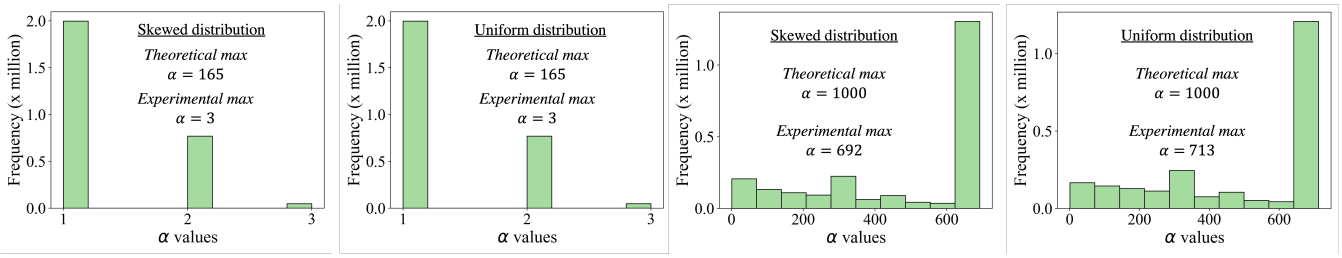
Changing D: This experiment studies the performance variation of Waffle while increasing the number of dummy objects, D, in the system. We increase the value of D from 20% of N to 100% of N, i.e., from 200k to 1M and Figure 3d highlights the results. As seen in the figure, the value of D has no significant effect on the system performance. The reason for this is that except for the binary search tree that maintains timestamps of dummy objects, no other data structure or logic in Algorithm 1 is affected by the size of dummy objects, and the algorithm does not cache these objects as well. Because of this, Waffle's performance remains independent of D. We set the default value of D to 330k to achieve high security by setting the two ratios of α (see Theorem 7.1) equal: $\frac{N-1}{B-R-f_D} = \frac{D}{f_D}$. The security analysis, which will be discussed in §8.3, indicated that the most common α value when the system is configured with the defaults used in this section is between 690 and 710; therefore we set $D = 700 * f_D = 350k$ as the default.

8.3 Security analysis

This section analyzes the security of Waffle experimentally for various system configurations and studies the security vs. performance trade-off. In particular, we evaluate the claim that Waffle

Expected security levels	Input Distribution	B	R	f_D	C (% of N)	D	Theoretical α	Observed max α	Theoretical β	Observed min β	Xput (ops/s)
High security	Skewed	10k	25	3914	99	4000	165	3	161	162	30
	Uniform	10k	25	3914	99	4000	165	3	161	162	28
Medium security	Skewed	2.5k	1k	500	2	350k	1000	692	5	9	10.8k
	Uniform	2.5k	1k	500	2	350k	1000	713	5	9	11.2k
Low security	Skewed	2.5k	2k	500	2	350k	999999	–	4	–	21.7k
	Uniform	2.5k	2k	500	2	350k	999999	–	4	–	22.4k

Table 2: Details of system parameters, the expected and observed α, β values, and throughput for various security levels.



(a) High security: skewed distribution **(b) High security: uniform distribution** **(c) Medium security: skewed distribution** **(d) Medium security: uniform distribution**

Figure 4: The histograms of the α values visible to the adversary for two different security levels and two extreme distributions. For a given security level, obliviousness stems from the similarity in the histograms for different input distributions.

is α, β-uniform by first calculating the expected α and β values using Theorems 7.1 and 7.2 and then measuring the actual α, β values for ***each*** server request to verify the bounds. The values of α and β depend on the system parameters defined in Table 1 (§7). Moreover, unlike Pancake, Waffle's security claims to handle correlated queries and not just independently drawn queries. Therefore, this section first analyzes Waffle's security when queries are independently drawn to study the performance-security trade-off (§8.3.1) and then analyzes Waffle's behavior when client queries are correlated (§8.3.2).

8.3.1 ***Independent queries.*** In analyzing Waffle's security vs. performance trade-off, we choose three sets of parameters yielding high, medium, and low security and observe (i) the corresponding α, β bounds and (ii) the performance for each. Note that as shown in Theorem 5.1, the lower the value of α and the higher the value of β, the higher is the system's security.

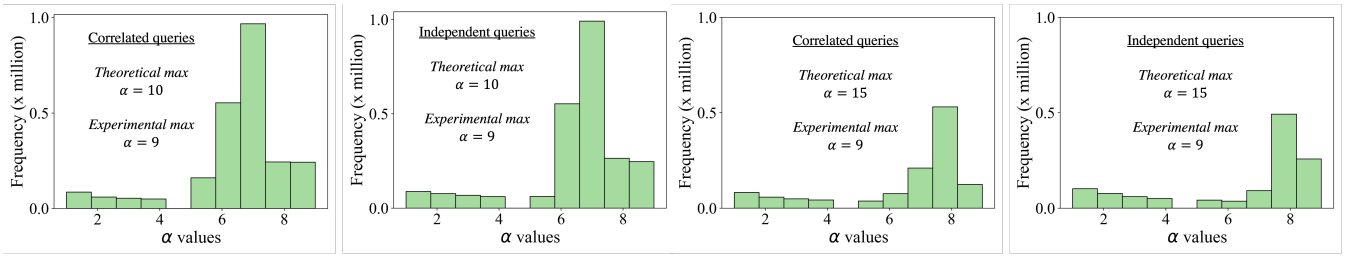
Each experiment identifies parameters that yield different expected security levels and executes Waffle by generating ~2.5 million requests drawn from two extreme distributions: a highly skewed distribution with a Zipf value of 0.99 and a uniform distribution. For each security and skewness level, we first show that Waffle maintains the α, β bounds, indicating that the sequence generated by Waffle is α, β-uniform, and then show for each security level how Waffle ensures obliviousness regardless of input distributions.

Recall that α is an upper bound dictating the maximum number of server accesses (Waffle accesses in batches) from when an object is written to when it is read next, whereas β is a lower bound dictating the minimum number of server accesses between reading an object and subsequently writing it back. Because Waffle uses a non-static assignment of plaintext keys to encoded keys, which changes *each time an object is read and written back to the server*, an adversary can only observe the α values and not the β values (between the read and a subsequent write, the PRF would have changed). Therefore, this experiment plots the histograms of an adversary-observable α values in Figures 4a to 4d. Meanwhile, Table 2 tabulates the various system parameters, the expected (or theoretical) and observed α, β values, and the resulting throughput. We discuss each of the results in detail below.

High security: To identify the parameters that can yield high security, we deployed both an exhaustive grid and a random parameter search techniques [33] that converged on parameter values with the highest $\frac{\beta}{\alpha}$ value (because security increases for higher β and lower α values). The grid search approach considers all (realistically) possible values of each varying parameter and picks the ones that yield the highest $\frac{\beta}{\alpha}$ value; whereas a random search technique randomly picks parameter values for a preset number of runs and identifies the values with the highest $\frac{\beta}{\alpha}$ value. Table 2 lists the system configurations that result in high security, i.e., low α and high β values, while still yielding a non-zero throughput. Maximizing β requires a large cache (in this case, we set it at $0.99 * N$) and minimizing α requires R to be much smaller compared to B.

Figures 4a and 4b plot the histograms of the number of requests that incurred unique α values for skewed and uniform input distributions, respectively. As seen in the graphs, while the theoretically expected value is $\alpha = 165$ (which is independent of the input distribution), the observed maximum is only 3 for both distributions. Meanwhile, as shown in Table 2, the expected β value, a lower bound, is 161 whereas the observed minimum is 162 for both distributions. These results prove that for the high security setting, Waffle generates α, β-uniform accesses.

A data system is oblivious if for any input sequence, it produces a uniform output sequence because an adversary observing the output sequence cannot distinguish between the differences in the



(a) R=20%: correlated queries **(b) R=20%: independent queries** **(c) R=40%: correlated queries** **(d) R=40%: independent queries**

Figure 5: The histograms of the α values visible to the adversary for R=20% & 40% of B when queries are correlated vs. independently drawn.

input sequence. Based on this, Waffle provides obliviousness for the high security level due to the fact that the adversary-observable α values are indistinguishable for both distributions, as seen in Figures 4a and 4b, with an average difference of 1,994 in the histogram buckets (out of ~2.5M requests). However, the performance for this setting as shown in Table 2, which is ~ 30 ops/s, as well as the parameter values such as $C = 0.99 * N$ indicates that this configuration is impractical.

Medium security: This security level uses the default parameters set in §8.2, as shown in Table 2. As noted in Table 2 as well as Figures 4c and 4d, the expected α value is 1000 and the observed values are 692 and 713 for skewed and uniform distributions, respectively. Similarly, the observed minimum β value of 9 is higher than the expected minimum of 5. Similar to the high security level, this highlights that Waffle guarantees α, β bounds, thus producing α, β-uniform server accesses. With regard to obliviousness, the adversary-observable α value histograms in Figures 4c and 4d differ marginally, with the average difference across different frequency buckets of 25,024 (out of ~ 2.5 million requests). Since only 1% of the requests differ in their αs, this indicates that the output α distribution remains fairly consistent for any input distribution. This combined with Waffle's design of changing the PRFs of plaintext keys after each accesses, implies that this security level maintains obliviousness. Moreover, this configuration yields a much higher throughput of ~ 11k ops/s.

Low security: This experiment sets system parameters which produce the highest throughput observed in §8.2, i.e. primarily by setting $R = 0.8 * B - 1$. This results in an extremely high expected α of 999,999 (given $N = 1M$). Although in terms of performance, this configuration executes ~22k ops/s, this configuration is *not oblivious*. This is because in the ~2.5 million requests served by the proxy in this experiment (in nearly 1200 batches of 2k real requests each), the proxy accesses only about 1200 real objects as fake queries since when $R = 0.8 * B - 1$ and $f_D = 0.2 * B$, f_R becomes 1. f_R serves as the primary indicator of the obliviousness by ensuring that even objects that are never requested by a client get accessed by the proxy. However, since this configuration produces such few fake queries on real objects, the unpopular objects can reside on the server for extended periods of time (precisely up to 999,999 server accesses) before the proxy reads them. The adversary can exploit this information to perform access pattern attacks. Hence, this configuration is not oblivious, in spite of producing high throughput. Note that we do not report α or β values or the histograms for this experiment since many real objects remained un-accessed, whose α and β values were unknown.

8.3.2 ***Correlated queries.*** Recall from §2 that Pancake's security requires client queries to be drawn independently. IHOP [38] performed a security attack on Pancake highlighting that for small datasets, an attacker can recover plaintext data when the input queries are correlated. IHOP used a real-world Wikipedia Clickstream [55] dataset that captures traversals across articles, which exhibit a correlated pattern. IHOP identified top 500 articles in different categories and measured the traversal probabilities between these articles to generate correlated queries. IHOP's attack accuracy is the highest for articles related to 'privacy' (i.e., articles on privacy) wherein 500,000 correlated queries request 500 unique articles (i.e., objects). Hence, we use this dataset and the corresponding correlated query workload in analyzing Waffle's security for correlated queries. Note that attack accuracy is higher for smaller datasets, hence we adhere to IHOP's attack setup.

In analyzing the security of Waffle for correlated queries, this experiment measures α values – the only measure visible to an adversary – of each server request under two settings: (i) when client queries have correlations (trace obtained from IHOP), and (ii) when client queries are independent (obtained by randomizing the correlated queries trace). Figure 5 plots the histograms of the α values for correlated and independently drawn queries for two different R values: R=20 (20% of B) and R=40 (40% of B). The other parameters remain unchanged across the experiments, with N=500: B=100, f_D=20 (20% of B), C=2%, and D=200 (proportionally similar to the defaults of §8.2). We chose to run the experiment with two different R values to highlight the security and performance trade-off for the correlated query experiment.

As seen in Figures 5a and 5b, with R=20, the α values observed by an adversary differ in ~0.8% of the requests (19,445 out of 2.4M server requests) when the queries are correlated vs. independently drawn. R=20 yields a throughput of 8.3 kops/s. Meanwhile, for R=40, the α values differ for ~3% of the requests (39164 out of 1.2M server requests) as shown in Figures 5c and 5d. R=40 yields a higher throughput of 15.2 kops/s. However, R=40 has lower security than R=20 because a higher percentage of requests differ in their α values. Conceptually, lower R leads to higher f_R, which directly impacts security. Interestingly, for the medium security experiment in §8.3.1 wherein queries are independently drawn from different distributions, R=40% of B caused only 1% of the requests to differ unlike the 3% in correlated queries.

This experiment leads to two conclusions: (i) Waffle can easily adapt to handle correlated queries, and (ii) the system parameters for high security may differ when queries are correlated vs. independently drawn, with the correlated query parameters likely yielding lower performance. This experiment also provides evidence that Waffle can be extended handle multi-maps (which can help with maintaining relational data).

8.4 Choosing system parameters

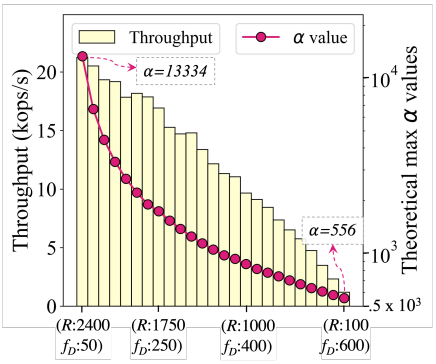


Figure 6: Security (measured using the theoretical max α values) vs. performance for various R and f_D values.

The conclusions from the correlated queries experiment leads to an important question of how to choose the system parameters. Identifying parameters that will yield high security is relatively straightforward by using either an exhaustive grid search or a random search technique, as explained in §8.3.1. However, this will likely result in unacceptably low performance. A better alternative is to start with a set of parameters that provide acceptable performance (such as R=40% of B and f_D=20% of B, as in the defaults of §8.2). The application can then perform the security analysis by measuring the α values on either a synthetic sample workload or any historical workloads. Note that since the security analysis only requires object keys and not values, it can be performed on local low-storage machines prior to offloading the database. The application can then iteratively fine-tune the parameters that will result in the desired security. Even after deploying, an application can monitor the α values observable to an adversary and can fine-tune parameters such as B, R, f_D, or C.

Since §8.3.1 only provides three levels of security and the corresponding performance trade-off, we conduct an experiment that gradually improves security (by reducing the theoretical α values) and measure the performance, as shown in Figure 6. Note that practically observed α values tend to be lower than the theoretical values, as seen in §8.3. This experiment changes R and f_D parameters, while retaining the default values (§8.2) for the other parameters. We only vary R and f_D for this analysis because these two parameters have the highest impact on performance (see Figures 3b and 3c) and the goal of this experiment is to study the performance-security trade-off. As seen in Figure 6, lower values of α, which indicate higher security, entails low performance. This trade-off allows an application to choose parameter values of R and f_D, which have the highest impact of performance, that can strike the desired balance between security and performance.

9 RELATED WORK

ORAM-based datastores: To-date, the most popular approach to ensure obliviousness is to use ORAM. A plethora of datastore designs employ ORAM to build an oblivious database including [2, 8, 11, 34, 48, 50, 51]. However, in spite of gaining much popularity within the research community, ORAM datastores are yet to be adopted by industry to build practical data systems. The primary reason is the prohibitive overhead incurred by ORAM datastores, as shown in many recently established lower bound studies [4, 6, 30, 31, 40, 41, 54]. The lower bound implies that for a database with N objects, each client request accessing a single object incurs $\Omega(logN)$ bandwidth overhead. A $\Omega(logN)$ bandwidth overhead can be unacceptable to support the scale of existing applications that serve tens of millions of requests [1, 5, 13, 52]. Because ORAM can handle active adversaries who can inject queries, hiding access patterns in such cases inevitably incurs bandwidth or storage overheads that depend on N.

Frequency-hiding datastores: Pancake [17], discussed in detail in §2, achieves obliviousness under a *passive- persistent adversary* by smoothening the access frequencies of all outsourced objects. A passive-persistent adversary only observes the access patterns on the server and can choose the distribution from which client queries are sent but it cannot inject individual queries, like in ORAM. Since we discuss how Waffle and Pancake compare throughout the paper, we focus on other works here.

Mavroforakis et al.[35] assume a similar but slightly weaker security model than Pancake and add fake queries based on the *a priori* known access distribution (either uniform or periodic). While the scheme presented in [35] can adapt to changing distributions, similar to Pancake, learning the new distribution is vital for the security of the system. Waffle on the other hand assumes nothing about knowing or learning users' access distribution. Sepehri [49] et al. apply the idea of frequency hiding to obfuscate *query patterns*, i.e., the keywords searched by users. The security of their scheme holds if the client queries are drawn from a Zipf distribution, unlike Waffle's security which can hold for any sequence of accesses.

10 CONCLUSION

This work presents Waffle, a datastore that protects data access patterns from a passive persistent adversaries. Waffle makes four major contributions: (i) it provides ***online obliviousness*** by adapting to any input sequence and obfuscates server access sequences, (ii) it empowers an application owner with more control over the data system and allows tuning security in exchange for performance, (iii) it uses a bounded cache, and (iv) it incurs constant bandwidth overhead. Waffle is the first system to hide access sequences under a passive persistent adversarial model. The evaluation of Waffle indicate that it performs **45-57%** better than Pancake, a state-of-the-art oblivious datastore, when its system parameters mirror that of Pancake; Waffle outperforms a concurrent-ORAM system, TaoStore, by **102x**, whereas an insecure datastore performs **5-6x** better than Waffle. We also present experimental security analysis to highlight the tunable property of Waffle and prove experimentally that Waffle can handle correlated queries while maintaining obliviousness. As future work, we aim to extend Waffle to add features including scalability, fault tolerance, and handling relational data.

REFERENCES

[1] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems* (2012), pp. 53–64.

[2] Bindschaedler, V., Naveed, M., Pan, X., Wang, X., and Huang, Y. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 837–849.

[3] Blackstone, L., Kamara, S., and Moataz, T. Revisiting leakage abuse attacks. *Cryptology ePrint Archive* (2019).

[4] Boyle, E., and Naor, M. Is there an oblivious ram lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science* (2016), pp. 357–368.

[5] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., et al. Tao: Facebook's distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIXATC 13)* (2013), pp. 49–60.

[6] Cash, D., Drucker, A., and Hoover, A. A lower bound for one-round oblivious ram. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18* (2020), Springer, pp. 457–485.

[7] Cash, D., Grubbs, P., Perry, J., and Ristenpart, T. Leakage-abuse attacks against searchable encryption. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security* (2015), pp. 668–679.

[8] Chakraborti, A., and Sion, R. Concuroram: High-throughput stateless parallel multi-client oram. *arXiv preprint arXiv:1811.04366* (2018).

[9] Cloud Adoption Statistics. https://bit.ly/3ZdCzpt. Accessed Feb 10, 2023.

[10] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), pp. 143–154.

[11] Crooks, N., Burke, M., Cecchetti, E., Harel, S., Agarwal, R., and Alvisi, L. Obladi: Oblivious serializable transactions in the cloud. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (2018), pp. 727–743.

[12] Dauterman, E., Fang, V., Demertzis, I., Crooks, N., and Popa, R. A. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (2021), pp. 655–671.

[13] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review 41*, 6 (2007), 205–220.

[14] Demertzis, I., Papadopoulos, D., Papamanthou, C., and Shintre, S. Seal: Attack mitigation for encrypted databases via adjustable leakage. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 2433–2450.

[15] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003), pp. 29–43.

[16] Goldreich, O., and Ostrovsky, R. Software protection and simulation on oblivious rams. *J. ACM 43*, 3 (May 1996), 431–473.

[17] Grubbs, P., Khandelwal, A., Lacharité, M.-S., Brown, L., Li, L., Agarwal, R., and Ristenpart, T. Pancake: Frequency smoothing for encrypted data stores. In *29th USENIX Security Symposium (USENIX Security 20)* (2020), pp. 2451–2468.

[18] Grubbs, P., Lacharité, M.-S., Minaud, B., and Paterson, K. G. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 315–331.

[19] Grubbs, P., Lacharité, M.-S., Minaud, B., and Paterson, K. G. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1067–1083.

[20] Gui, Z., Johnson, O., and Warinschi, B. Encrypted databases: New volume attacks against range queries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 361–378.

[21] Health Care Cloud Computing Trends. https://bit.ly/3ZyCWdI. Accessed Feb 10, 2023.

[22] Herlihy, M. P., and Wing, J. M. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[23] Islam, M. S., Kuzu, M., and Kantarcioglu, M. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *Ndss* (2012), vol. 20, Citeseer, p. 12.

[24] Kellaris, G., Kollios, G., Nissim, K., and O'neill, A. Generic attacks on secure outsourced databases. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 1329–1340.

[25] Kornaropoulos, E. M., Papamanthou, C., and Tamassia, R. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1033–1050.

[26] Lacharité, M.-S., Minaud, B., and Paterson, K. G. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 297–314.

[27] Lacharité, M.-S., and Paterson, K. G. A note on the optimality of frequency analysis vs. ℓ_p-optimization. *Cryptology ePrint Archive* (2015).

[28] Lacharité, M.-S., and Paterson, K. G. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *Cryptology ePrint Archive* (2017).

[29] Lamport, L. The part-time parliament. In *Transactions on Computer Systems*. ACM, 1998, pp. 133–169.

[30] Larsen, K. G., Malkin, T., Weinstein, O., and Yeo, K. Lower bounds for oblivious near-neighbor search. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2020), SIAM, pp. 1116–1134.

[31] Larsen, K. G., and Nielsen, J. B. Yes, there is an oblivious ram lower bound! In *Annual International Cryptology Conference* (2018), Springer, pp. 523–542.

[32] Li, J., Qin, C., Lee, P. P., and Zhang, X. Information leakage in encrypted deduplication via frequency analysis. In *2017 47th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)* (2017), IEEE, pp. 1–12.

[33] Liashchynskyi, P., and Liashchynskyi, P. Grid search, random search, genetic algorithm: a big comparison for nas. *arXiv preprint arXiv:1912.06059* (2019).

[34] Maiyya, S., Ibrahim, S., Scarberry, C., Agrawal, D., Abbadi, A. E., Lin, H., Tessaro, S., and Zakhary, V. QuORAM: A Quorum-Replicated fault tolerant ORAM datastore. In *31st USENIX Security Symposium (USENIX Security 22)* (Boston, MA, Aug. 2022), USENIX Association, pp. 3665–3682.

[35] Mavroforakis, C., Chenette, N., O'Neill, A., Kollios, G., and Canetti, R. Modular order-preserving encryption, revisited. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (2015), pp. 763–777.

[36] Naveed, M., Kamara, S., and Wright, C. V. Inference attacks on property-preserving encrypted databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), pp. 644–655.

[37] Oya, S., and Kerschbaum, F. Hiding the access pattern is not enough: Exploiting search pattern leakage in searchable encryption. In *USENIX Security Symposium* (2021), pp. 127–142.

[38] Oya, S., and Kerschbaum, F. Ihop: Improved statistical query recovery against searchable symmetric encryption through quadratic optimization. In *31st USENIX Security Symposium (USENIX Security 22)* (2022), pp. 2407–2424.

[39] Papadimitriou, A., Bhagwan, R., Chandran, N., Ramjee, R., Haeberlen, A., Singh, H., Modi, A., and Badrinarayanan, S. Big data analytics over encrypted datasets with seabed. In *OSDI* (2016), vol. 16, pp. 587–602.

[40] Patel, S., Persiano, G., and Yeo, K. What storage access privacy is achievable with small overhead? In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems* (2019), pp. 182–199.

[41] Persiano, G., and Yeo, K. Lower bounds for differentially private rams. In *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part I 38* (2019), Springer, pp. 404–434.

[42] Petersen, C., Simonsen, J. G., and Lioma, C. Power law distributions in information retrieval. *ACM Transactions on Information Systems (TOIS) 34*, 2 (2016), 1–37.

[43] Poddar, R., Boelter, T., and Popa, R. A. Arx: an encrypted database using semantically secure encryption. *Cryptology ePrint Archive* (2016).

[44] Poddar, R., Wang, S., Lu, J., and Popa, R. A. Practical volume-based attacks on encrypted databases. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)* (2020), IEEE, pp. 354–369.

[45] Popa, R. A., Redfield, C. M., Zeldovich, N., and Balakrishnan, H. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the twenty-third ACM symposium on operating systems principles* (2011), pp. 85–100.

[46] Redis. https://redis.io/. Accessed Feb 10, 2023.

[47] Ren, L., Fletcher, C., Kwon, A., Stefanov, E., Shi, E., Van Dijk, M., and Devadas, S. Constants count: Practical improvements to oblivious ram. In *24th USENIX Security Symposium (USENIX Security 15)* (2015), pp. 415–430.

[48] Sahin, C., Zakhary, V., El Abbadi, A., Lin, H., and Tessaro, S. Taostore: Overcoming asynchronicity in oblivious data storage. In *2016 IEEE Symposium on Security and Privacy (SP)* (2016), IEEE, pp. 198–217.

[49] Sepehri, M., and Kerschbaum, F. Low-cost hiding of the query pattern. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security* (2021), pp. 593–603.

[50] Stefanov, E., and Shi, E. Oblivistore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 253–267.

[51] Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., and Devadas, S. Path oram: an extremely simple oblivious ram protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 299–310.

[52] The Infrastructure Behind Twitter: Scale. https://bit.ly/3KJR7J1. Accessed Feb 10, 2023.

[53] TLS. https://datatracker.ietf.org/doc/html/rfc5246. Accessed July 14, 2023.

[54] Weiss, M., and Wichs, D. Is there an oblivious ram lower bound for online

reads? *Journal of Cryptology 34*, 3 (2021), 18.

[55] Wikipedia ClickStream Dataset. https://dumps.wikimedia.org/other/clickstream/. Accessed July 14, 2023.

[56] Zhang, Y., Katz, J., and Papamanthou, C. All your queries are belong to us: The power of file-injection attacks on searchable encryption. In *USENIX Security Symposium* (2016), vol. 2016, pp. 707–720.