PANCAKE: Frequency Smoothing for Encrypted Data Stores

Paul Grubbs* Cornell Tech
Anurag Khandelwal* Yale University
Marie-Sarah Lacharité*† Royal Holloway, University of London
Lloyd Brown UC Berkeley
Lucy Li Cornell Tech
Rachit Agarwal Cornell University
Thomas Ristenpart Cornell Tech

Abstract

We present PANCAKE, the first system to protect key-value stores from access pattern leakage attacks with small constant factor bandwidth overhead. PANCAKE uses a new approach, that we call *frequency smoothing*, to transform plaintext accesses into uniformly distributed encrypted accesses to an encrypted data store. We show that frequency smoothing prevents access pattern leakage attacks by passive persistent adversaries in a new formal security model. We integrate PANCAKE into three key-value stores used in production clusters, and demonstrate its practicality: on standard benchmarks, PANCAKE achieves 229× better throughput than non-recursive Path ORAM — within 3–6× of insecure baselines for these key-value stores.

1 Introduction

High-performance data stores, such as key-value stores [1, 19, 32], document stores [43], and graph stores [33, 47], are a building block for many applications. For ease of management and scalability, many organizations have recently transitioned from on-premise to cloud-hosted data stores (e.g., [19]), and from server-attached to disaggregated storage [21, 28, 35, 65]. While beneficial, these deployment settings lead to significant security concerns: data accesses that used to be contained within a trusted domain (an organization's premises or within a server) are now visible to potentially untrusted entities.

A now-long line of work has shown that, even if the data is encrypted, the observed data access patterns can be exploited to learn damaging information about the data, through access pattern attacks such as frequency analysis (e.g., [12, 26, 29, 31, 37]). These attacks require only a passive persistent adversary, that is, one that observes access patterns but does not actively performs accesses. Existing techniques that are secure against access pattern attacks, such as oblivious RAMs [23], target stronger security models where the adversary can actively perform data accesses; as we discuss in detail in §2, these techniques have fundamental performance overheads [10, 39, 40, 49, 50, 66] making them impractical for most settings. Thus, the problem of building high-performance data stores that are secure against access pattern attacks by persistent passive adversaries remains open.

We make three core contributions towards resolving this open problem. First, we introduce a formal security model that captures (just) passive persistent adversaries in encrypted data store settings. Specifically, we model honest users' queries to the data store as a sequence of data access requests sampled from a time-varying distribution. The encryption mechanism can obtain an estimate of the distribution; the adversary both knows the distribution and obtains the transcript of (encrypted) queries and responses. Informally, we say that a mechanism is secure if the adversary is unable to distinguish the transcript from a sequence of uniformly distributed accesses to random bit strings. We capture this security goal in what we call real-or-random indistinguishability under chosen dynamic-distribution attack (ROR-CDDA).

Our second contribution is *frequency smoothing*, a mechanism that is ROR-CDDA secure, that is, provides security against access pattern attacks by passive persistent adversaries. The key insight underlying frequency smoothing is that, for passive persistent adversaries, data access requests being chosen from a distribution provides a source of "uncertainty" that can be leveraged in a principled manner. For instance, if requests were sampled from a uniform distribution, it is easy to see that the adversary gains no additional information from observing accesses patterns. However, most real world distributions are not uniform. Frequency smoothing uses the estimate of the data access distribution to transform a sequence of requests into uniform accesses over encrypted objects (hereafter, key-value pairs) in the data store.

Frequency smoothing carefully combines four techniques: selective replication, fake accesses, batching of queries, and dynamic adaptation. Selective replication creates "replicas" of key-value pairs that have high access probability relative to others in the data store. This serves to partially smooth the distribution over (replicated) key-value pairs. For the remaining non-uniformity, we combine selective replication with the idea of "fake" queries [42]. These are sampled from a carefully crafted fake access distribution to boost the likelihood of accessing replicated key-value pairs until the resulting distribution is entirely uniform. Security requires ensuring that fake and real queries be indistinguishable; to achieve this, we issue small batches of encrypted queries, where each query is either real or fake with equal probability. Finally, we show how one can dynamically adapt to changes in the underlying data access distribution by opportunistically adapting the replica creation as well as the fake access distribution.

Our third contribution is the design, implementation, and evaluation of an end-to-end system — PANCAKE— that realizes frequency smoothing, and can be used with existing data stores. PANCAKE builds upon the encryption proxy system model used in many deployment settings, where a proxy

*The first three authors contributed equally to the work.
†Portions of this work were done while visiting Cornell University.

acts as an intermediary between clients and the data store. PANCAKE uses this proxy to maintain an estimate of the time-varying access distribution (based on incoming requests from the clients), as well as securely execute read/write queries by using pseudorandom functions for keys and authenticated encryption for values. Assuming the distribution estimates are sufficiently good (we make this precise in §4), PANCAKE achieves ROR-CDDA security.

We analyze PANCAKE's performance both analytically and empirically. Specifically, we show that PANCAKE's server-side storage and bandwidth overheads are within a constant factor of insecure data stores; while the proxy storage can be large in the worst-case (depending on the underlying data access distributions), our empirical evaluation demonstrates minimal overheads for heavy-tailed, real-world distributions.

We integrate PANCAKE with two key-value stores used in production clusters — a main-memory based key-value store Redis [54] and an SSD-based key-value store RocksDB [55]. Evaluation over a variety of workloads demonstrates that PANCAKE consistently achieves throughput within $3-6\times$ of the respective key-value store that does not protect against access pattern leakage attacks. Sensitivity analysis against various workloads, deployment scenarios (within a cloud and across wide-area networks), query loads, and more, demonstrates that PANCAKE maintains its performance across a diversity of evaluated contexts. We also compare PANCAKE performance against Path ORAM [63], a representative system from the ORAM literature. Across various workloads, PANCAKE achieves significantly better throughput (sometimes by as much as $229\times$) than PathORAM. Of course, ORAMs are designed to prevent a broader range of attacks (e.g., active injection attacks); our comparison should be interpreted as highlighting the huge efficiency gap between countermeasures in the two threat models. An end-to-end implementation of PANCAKE along with all the details to reproduce our results is available at `https://github.com/pancake-security`.

PANCAKE is a first step toward designing high-performance data stores that are secure against access pattern attacks by passive persistent adversaries. We outline limitations, open research questions, and future research avenues in §7.

2 The PANCAKE Security Model

We introduce a new security model for capturing passive persistent attacks against encrypted data stores. We also discuss prior approaches for resisting access pattern attacks.

System model. We focus on key-value (KV) stores that support (single-key) get, put, and delete operations on KV pairs (k,v) submitted by one or more clients. Our results can, however, be applied to any data store that supports read/write/delete operations.

We consider outsourced storage settings where one or more clients want to utilize a KV store securely. PANCAKE employs a proxy architecture commonly used by encrypted data stores in practice [15, 45, 51, 60] and in the academic literature [53, 57, 62]. This deployment setting assumes multiple client applications route query requests through a single trusted proxy. The proxy manages the execution of these queries on behalf of the clients, sending queries to some storage service. Our security model and results apply equally well to a setting with a single client and no proxy.

We assume all communication channels are encrypted, e.g., using TLS. This does not prevent the storage service from seeing requests. The proxy therefore encrypts each KV pair (k,v) by applying a pseudorandom function (PRF) to the key, denoted $F(k)$, and symmetrically encrypting the value, denoted $E(v)$. We assume that the values are all the same size, perhaps via padding —i.e., there is no length leakage. The secret keys needed for F and E are stored at the proxy. Because F is deterministic, the proxy can perform operations for key k by instead requesting $F(k)$. This standard approach is used in a variety of commercial products [5, 15, 45, 51, 60].

Security model. Our security model captures passive persistent adversaries in such encrypted data store settings. The adversary observes all (encrypted) accesses but does not actively perform its own (e.g., via a compromised client).

We model honest client requests as queries sampled from a distribution π over keys: for each key k, the probability of a query (get, put, or delete) on that key is denoted $\pi(k)$. The distribution may change over time. While we primarily focus on the case where queries are independent draws from π, we discuss correlated queries and how this relates to ORAM security in the full version [25].

In our model, the adversary does not have access to cryptographic keys, but can observe all encrypted queries to, and corresponding responses from, the storage server. The adversary does not change the client queries, the responses, or the stored data. The adversary knows π, but the random draws from π that constitute individual accesses are (initially) hidden. The adversary wins if it can infer *any* information about the resulting sequence of accessed plaintext KV pairs; we formalize this further in §4.3. We do not target hiding the time at which a query is made; fully obfuscating timing requires a constant stream of accesses to the data store, which is prohibitively expensive in many contexts. (Our approaches can nevertheless be extended to hide timing in this way.) See §7 for more discussion on the limitations of our security model.

Access pattern attacks. Without further mechanisms, the basic PRF and encryption approach leaks the pattern of accesses to the adversary. In various contexts an attacker can combine this leakage with knowledge about π [8, 12, 29, 46] to mount damaging attacks like frequency analysis: order the KV pairs by decreasing likelihood of being accessed $k_1, k_2, \ldots$, and guess that the most frequently accessed encrypted value is k_1, the second most frequently accessed is k_2, etc. In general, in our security model the adversary can use knowledge of the distribution π to:

- infer key identities,
- identify when specific keys are accessed, and,
- detect and identify changes in key popularities over time.

Our goal is to protect against such access pattern attacks.

Prior approaches. Access pattern and related attacks have been treated in the literature before; we briefly overview three lines of work related to our results.

Oblivious RAMs (ORAMs): Existing ORAM designs provide security against access pattern attacks even in settings where the adversary can actively inject its own queries. The core challenge with ORAM based approaches is their overheads — several recent results [10, 39, 40, 49, 50, 66] have established strong lower bounds on ORAM overheads: for a data store with n key-value pairs, any ORAM design must either: (1) use constant proxy storage but incur $\Omega(\log n)$ bandwidth overheads; or, (2) must use $\Theta(n)$ storage at the proxy and incur constant bandwidth overheads. Unfortunately, both of these design points are inefficient for data stores that store billions of key-value pairs [4, 11, 20, 24, 64]. At such a scale, $\Omega(\log n)$ bandwidth overheads result in orders-of-magnitude throughput reduction [14]. State-of-the-art ORAM designs that achieve constant bandwidth overheads in theory [3] have large constants hidden within the asymptotic result (as much as 2^{100} [3]), resulting in high concrete overheads. For many applications, ORAM overheads may be unacceptable.

Snapshot attacks: Another recent line of work has targeted what's called a snapshot threat model, where the adversary does not persistently observe queries and only obtains a one-time copy (snapshot) of the encrypted data store [38, 48, 52]. One of these [38] propose frequency-smoothed encryption, a technique similar to our selective replication mechanism. It, like other schemes built for the snapshot model, does not resist attacks by passive persistent adversaries. Further, the snapshot model is unrealistic for existing storage systems [27].

Fake queries: Mavroforakis et al. [42] explore the idea of injecting fake queries to obfuscate access patterns in the context of range queries and (modular) order-preserving encryption. In a security model where boundaries between the queries are not known to the adversary, this can provide security albeit with high bandwidth overheads. However, if query boundaries are known to the adversary (as in our model and in practice), the adversary can trivially distinguish between real and fake queries because the last query sent is always the real one. That said, our work uses the idea of fake queries from [42], adapting it to our KV store setting (see §4.2) and combining it with further techniques to ensure security.

3 PANCAKE Overview

We now provide a brief overview of PANCAKE's core technique — frequency smoothing. We relegate the discussion of PANCAKE's design details to subsequent sections.

Frequency smoothing. Most data stores already gather statistics about data access patterns for load balancing, debugging and performance tuning [2, 4]. PANCAKE's design exploits that all clients route their queries via the proxy; thus, the proxy can learn information about the frequency of plaintext accesses. We provide intuition on frequency smoothing technique assuming perfect estimates, i.e., $\hat{\pi} = \pi$. We start with the case that π does not change over time, and discuss the dynamic case at the end of the section.

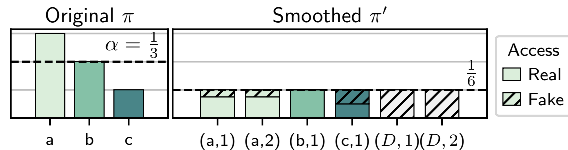


Figure 1: **Frequency smoothing example.** (Left) Original distribution over keys. (Right) Distribution over replicas after frequency smoothing. Ratio of real to fake accesses is the ratio of their areas.

PANCAKE uses the estimate of π to perform frequency smoothing. The key technical challenge is how to efficiently transform accesses distributed according to π over (plaintext) keys to a uniform distribution over encrypted keys. PANCAKE achieves this through a combination of three techniques: *selective replication*, *fake queries*, and *batching*. In fact, either selective replication or fake queries along with batching could be used to smooth frequency, but with prohibitive performance overheads as we explain below. The trick will be combining the three together in order to achieve an efficient solution.

Selective replication creates a number of copies of a key k (called *replicas*[1]) proportional to their likelihood of access: the more likely, the more replicas. When accessing a key, one of its replicas is chosen at random. Theoretically, a value α can be selected such that each $\pi(k) = R(k) \cdot \alpha$ for some integer $R(k)$. Key k would get $R(k)$ replicas. This would smooth the distribution to uniform. However, it leads to impractical storage overheads for typical distributions — the overhead for the YCSB workload (§6) would be $15\times$.

Instead, PANCAKE creates $R(k)$ replicas of k, just enough to ensure only that $\pi(k)/R(k) \leq 1/n$ where n is the number of items in the data store. We refer to $\alpha = 1/n$ as the replica threshold. As we will show in §4.1, this ensures the total number of replicas n', although dependent on the distribution π itself, is always $\leq 2n$. Since an adversary may learn some distributional information from n', we add a dummy key D with $2n - n'$ replicas, so that the total number of replicas is always exactly $2n$, regardless of π. For example, given the distribution $(1/2, 1/3, 1/6)$ over $n = 3$ keys a, b, c and threshold $1/3$, selective replication creates two replicas of key a (denoted as (a, 1), (a, 2)), one replica each of b and c (denoted as (b, 1) and (c, 1), respectively) and two replicas for the dummy key D (denoted as (D, 1) and (D, 2)). Figure 1 plots the frequencies.

The resulting distribution over replicas is not quite uniform. In our example, the distribution over (a, 1), (a, 2), (b, 1), (c, 1), (D, 1), (D, 2) is $(1/4, 1/4, 1/3, 1/6, 0, 0)$. PANCAKE therefore

[1] We use the term *replica* to refer to both the original key and its copies.

uses an equal proportion of fake queries mixed in with real ones in order to ensure a uniform distribution over accesses. To do so, PANCAKE computes a complementary fake access distribution over replicas so that the sum of the probability of a fake access and real access for any given replica is equal to $1/2n$, where $2n$ is the total number of replicas. Every time an access is made, it is chosen to be either fake or real with probability $1/2$. In our example, using a fake access distribution of ($1/12$, $1/12$, 0, $1/6$, $1/3$, $1/3$) across the four replicas ensures each replica has a total access probability of exactly $1/6$. We will show that adding fake queries in this manner always ensures equal probability for any key being accessed.

To support updates to values, every access is a read followed by a write of a freshly encrypted value. For keys with many replicas, we cannot change all replicas immediately as this would leak that these encrypted values are linked. Instead PANCAKE updates one of the replicas, caches the new value, and opportunistically updates the remaining replicas using subsequent fake or real queries to the replica. This could require a large cache in the worst case, but we show empirically in §6 that the cache remains small for typical workloads.

To service a (real) query from a client, PANCAKE performs a sequence of B accesses randomly chosen from either the real or the fake distribution, inserting the actual request into one of those chosen to be real. There is a small chance that the client's request cannot be served, in which case PANCAKE puts the query into a queue until the next client request arrives. We show that with $B = 3$, PANCAKE can ensure delivery of client requests in a timely manner (we make this precise in the next section), while maintaining that the probability of accessing any sequence of B encrypted keys is equally likely.

One could achieve security without selective replication by increasing the ratio of fake queries to real queries, but a larger value of B will be needed to ensure client requests are not stalled for arbitrarily long time. This, in turn, would result in high bandwidth overheads for many distributions. Thus, the combination of selective replication and fake queries, as in frequency smoothing, is necessary to ensure small overheads. With our chosen parameters, we will prove a storage overhead of $2\times$ and a bandwidth overhead of $3\times$ of insecure KV stores, independent of the underlying distribution. Moreover, we will prove that PANCAKE's protocol is secure if the estimate $\hat{\pi}$ is sufficiently good.

Dynamic query distributions. To allow PANCAKE to maintain its security and performance guarantees even when access distributions change, we extend the above design using an efficient algorithm that dynamically updates the fake query probabilities and replica allocations across keys. Recall that the total number of replicas in PANCAKE is always $2n$, regardless of the distribution. This means that when the distribution changes, for every key that must lose a replica, another must gain a replica. Therefore, handling distribution changes simply requires reassigning replicas for all such key-pairs.

PANCAKE uses a specialized replica swapping protocol to efficiently adjust the allocation of replicas in parallel with servicing client requests. The key challenge is that a request must be serviced by one of the old replicas, not a newly allocated one, until all the new replicas have the appropriate value propagated to them. We show that we can temporarily lower the ratio of real to fake queries, which, combined with appropriate temporary caching of values during the transition, maintains the invariant that every access to the store is uniformly distributed, guaranteeing security (§5).

4 PANCAKE Design: Static Distribution Case

We now provide details on the design and implementation of PANCAKE. In this section, we focus on the case of a static distribution, and extend PANCAKE's design to efficiently handle dynamic changes in the next section. We start the section with the data storage (§4.1) and frequency smoothing (§4.2) mechanisms in PANCAKE, and then provide a formal security analysis for PANCAKE's design (§4.3). We close the section with performance analysis of PANCAKE's storage requirements and bandwidth overheads for query execution (§4.4).

4.1 Data Storage

PANCAKE is backward-compatible with existing data stores — it requires no modifications on how data is sharded across multiple cores or machines, and how queries are executed in the underlying data store. Thus, PANCAKE naturally benefits from the many properties of existing data store, e.g., elasticity, fault tolerance, data persistence, etc. The core of the PANCAKE design is a proxy, which we describe below.

The PANCAKE proxy. The main functionality of the PANCAKE proxy is to initialize the data store, to implement frequency smoothing, and to execute queries on behalf of clients (encryption/decryption of query requests/responses). The proxy maintains several data structures to achieve its functionalities:

- **Observed query distribution ($\hat{\pi}$):** The proxy maintains the probability of access for individual keys, based on the histogram of accesses across keys. This "observed" distribution is an estimate of the underlying distribution, and is also used to detect changes in distribution over time.
- **Fake query distribution (π_f):** The proxy also maintains a fake probability of access for each individual key. We will discuss below how the fake distribution is computed.
- **Key $\rightarrow$ replica counts:** PANCAKE's selective replication mechanism may create one or more replicas for KV pairs. The proxy maintains a map $k \rightarrow R(k)$ from keys to their number of replicas, for all keys with $R(k) > 1$.
- **UpdateCache:** To securely handle write queries, we use a data structure that stores a map $k \rightarrow (v, \texttt{UpdateMap})$, where `UpdateMap` is a bitmap of length $R(k)$ denoting whether or

not a particular replica of k has been updated or not. We provide more details below.

- **Query queue:** This stores outstanding client queries.

The rest of the section details how the PANCAKE proxy uses these data structures to realize its functionalities. But first we make two observations about proxy storage and scalability.

Regarding PANCAKE proxy storage requirements, we note that storing the probability for a key as floating-point values requires 8 bytes of storage per key; given that the size of values in many real-world applications is of the order of kilobytes [4], storing the real and the fake distributions requires a tiny fraction of the entire dataset size. For instance, with 4 kilobyte values, the fraction works out to a mere 0.39%. Similarly, the key $\rightarrow$ replica counts data structure is also tiny. The size of UpdateCache, on the other hand, depends on the query distribution as well as the write rates; we evaluate the UpdateCache size empirically for realistic workloads in §6.

The PANCAKE proxy is implemented to efficiently scale with multiple cores. For the multi-core implementation, the first four data structures are shared by all PANCAKE proxy cores, while each core maintains its own query queue (for queries "assigned" to that core). Our proxy implementation ensures high performance (highly concurrent read-write rates) for data structures shared across cores. The first three data structures are updated at coarse-grained timescales (e.g., due to significant changes in the query distributions) and thus, simple arrays suffice for our purposes. UpdateCache, on the other hand, requires concurrent read/write operations; to this end, our implementation uses a Cuckoo hashmap [41] that can support 40 million read/write operations per second on a commodity server.

4.2 Frequency Smoothing

We now describe PANCAKE's frequency smoothing techniques for static distributions, specifically the algorithms to initialize the data store (with selective replication) and execute queries (with real queries, fake queries, and batching).

Initializing the data store. PANCAKE transforms a plaintext data store $\mathsf{KV} = \{(k_i, v_i)\}$ with n KV pairs into a data store KV' with $n' \geq n$ encrypted KV pairs. At the same time, PANCAKE transforms accesses distributed according to π over the keys of KV to a sequence of uniform accesses over the encrypted keys of KV'. To distinguish between plaintext keys and encrypted ones, we refer to the latter as labels. PANCAKE use an estimate $\hat{\pi}$ of π. During initialization, $\hat{\pi}$ can be assumed to be uniform, and the techniques from §5 can later be used to transition to a more accurate estimate. Alternatively, in many settings one will provide a warm start by initializing PANCAKE with a $\hat{\pi}$ learned from performance or other logs.

In generating KV', we use selective replication to add replicas to KV' for keys accessed frequently according to $\hat{\pi}$. If we set a threshold α, then for each $(k,v) \in \mathsf{KV}$ we generate $R(k,\hat{\pi},\alpha) = \lceil \hat{\pi}(k)/\alpha \rceil$ replicas: key-value pairs $((k,j),v)$ where j ranges over 1 to $R(k,\hat{\pi},\alpha)$. When $\hat{\pi}$ and α are clear from context, we will omit them and simply write $R(k)$.

Each replica (k,i) is then protected by applying a secretly keyed pseudorandom function F (e.g., HMAC) to the replica identifier to generate a label $F(k,i)$. We apply authenticated encryption E to the value. Thus ultimately $\mathsf{KV}' = \{(F(k_i, j), E(v_i)\}$ for $1 \leq i \leq n$ and where $1 \leq j \leq R(k_i)$ for each i. For simplicity, we have omitted in our notation the two required cryptographic secret keys, and that we cryptographically bind labels and value ciphertexts together by using the label as associated data with E. A straightforward calculation shows that for any $\hat{\pi}$ and α, $n' \leq n + 1/\alpha$.

The second initialization task is to compute a fake distribution π_f over replicas. Here we adapt a technique from Mavroforakis et al. [42]. In particular we pick a constant $0 < \delta \leq 1$ (this choice is explained in more detail below) and then craft π_f so that the probability $p(k,j)$ of accessing any replica (k,j) is: (1) equal to $1/n'$ and (2) a convex combination of the probability of truly accessing a replica and performing a fake access. Namely we ensure that

$$p(k,j) = \delta \cdot \frac{\hat{\pi}(k)}{R(k)} + (1-\delta) \cdot \pi_f(k,j) = \frac{1}{n'} \,. \qquad (1)$$

This corresponds to the following randomized process. Flip a δ-biased coin. If it comes up heads, randomly choose a replica for some real query drawn according to π; otherwise, choose a replica to access according to the fake distribution π_f.

The constant δ must be chosen so that $\delta \leq R(k)/(n' \cdot \hat{\pi}(k))$ for every key k; otherwise, it may not be possible to assign valid (non-negative) probability $\pi_f(k,j)$ to satisfy Equation 1 for some key k. We use $\delta = 1/(n'\alpha)$, which is always valid.

Note that δ corresponds to the proportion of real queries: if α is set too high, then most queries would be fake. At the same time, since $n' \leq 1/\alpha + n$, setting α too low would cause KV' to grow too large. We set $\alpha = 1/n$ since it corresponds to a sweet spot: $n' \leq 2n$, i.e., KV' is at most twice as large as KV, and $\delta \geq 1/2$, i.e., at least half the queries are real.

Dummy replicas. We note that the approach outlined above would result in a different number of total replicas for different distributions (although upper-bounded by $2n$), which leaks information about the distribution. To avoid this leak, PANCAKE preemptively initializes KV' with enough *dummy replicas* so that the total number of replicas is always $2n$.

Dummy replicas are KV pairs $(F(D,j), E(D))$, for $j = 1, \ldots, 2n - n'$ (n' is the number of "real" replicas for $\hat{\pi}$), where the dummy key D is unique and does not exist in the original set of keys. Dummy replicas are accessed only with fake accesses; therefore, $\hat{\pi}(D) = 0$ and the fake access probability is $\pi_f(D) = \alpha/(2n\alpha - 1)$ (derived from Eq. 1). Note that since the total number of replicas is now $2n$, the proportion of real queries $\delta = 1/(2n\alpha) = 1/2$ for $\alpha = 1/n$.

A pseudocode description of PANCAKE's initialization (in-

Init($\hat{\pi}$, KV, α):

$n \leftarrow |\mathsf{KV}|$
$\mathsf{KV}' \leftarrow \emptyset$
$n' \leftarrow 0$
For $(k, v) \in \mathsf{KV}$:
  $R(k) \leftarrow \lceil \hat{\pi}(k)/\alpha \rceil$
  For $j \in [1, \ldots, R(k)]$:
    $\pi_f(k, j) \leftarrow \frac{\alpha - \hat{\pi}(k)/R(k)}{2n\alpha - 1}$
    $\mathsf{KV}' \xleftarrow{\cup} \{(F(k, j), E(v))\}$
  $n' \leftarrow n' + R(k)$
For $j \in \{1, \ldots, 2n - n'\}$:
  $\pi_f(D, j) \leftarrow \frac{\alpha}{2n\alpha - 1}$
  $\mathsf{KV}' \xleftarrow{\cup} \{(F(D, j), E(D))\}$
$\delta \leftarrow \frac{1}{2n\alpha}$
Return $\mathsf{KV}', \pi_f, R, \delta$

Batch(k):

$j \leftarrow\!\!\$ \{1, \ldots, R(k)\}$
AddToQueue(k, j)
For $i = 1$ to B:
  $q_{\text{type}} \leftarrow_{\delta} \{0, 1\}$
  If $q_{\text{type}} = 0$:
    $(k_i, j_i) \leftarrow\!\!\$ \pi_f$
  Else:
    If QueueNotEmpty:
      $(k_i, j_i) \leftarrow$ Dequeue()
    Else:
      $k_i \leftarrow\!\!\$ \hat{\pi}$
      $j_i \leftarrow\!\!\$ \{1, \ldots, R(k_i)\}$
$\ell \leftarrow \{F(k_1, j_1), \ldots, F(k_B, j_B)\}$
Return ℓ

Figure 2: **PANCAKE's initialization and batch access algorithms** for a plaintext data store KV, distribution estimate $\hat{\pi}$, and threshold α.

cluding dummy replicas) appears in Figure 2.

Query execution. Intuitively, we will follow the randomized process associated to Equation 1 to mix fake and real accesses. To increase the probability that a client's real access is handled right away, PANCAKE in fact sends a small batch of accesses to KV$'$ for each client request. In particular, when a client submits an access request for key $k \in \mathsf{KV}$, PANCAKE runs the Batch algorithm shown in Figure 2. It randomly chooses a replica j of k, adds (k, j) to the query queue, and prepares a batch of B accesses to KV$'$. By default we set $B = 3$ (we will justify our choice in §4.4). For each of these accesses, it samples a bit q_{type} according to δ that determines whether the access is real (heads) or fake (tails). For each q_{type} that comes up heads (real) in the batch we attempt to send a value from the query queue. If the query queue is empty, then the client simulates a real access by sampling a key from $\hat{\pi}$ itself (denoted $k \leftarrow\!\!\$ \hat{\pi}$) and choosing a replica at random. For each fake access, the client samples a replica according to π_f. The resulting batch of replicas have the pseudorandom function F applied before being sent to the server. Note that Batch imposes bandwidth overhead exactly $B\times$ over a KV store that just uses encryption and leaks access patterns.

Note that the batching done in the PANCAKE proxy does not require all queries in the batch to be sent to the same shard/server; the batching is completely independent of the sharding mechanism used on the server and queries in the batch are independently forwarded to respective shards. Upon retrieving the associated values, PANCAKE decrypts the ones requested by clients and returns them.

It is critical that PANCAKE only sends a single batch for each client request. If instead the proxy sent batches until the query queue was empty, frequency information about which keys clients access would leak. For example, if one uses $B = 1$ and kept submitting until the queue is empty, then the final access to KV$'$ must be a client request. Thus PANCAKE defers handling a query until a later batch if necessary, increasing latency. We show experimentally that for most loads this latency increase is acceptably low (§6.3). In practice PANCAKE can vary B as a function of load: decrease B at high load (to lower bandwidth overhead) and increase B at low load (to lower latency). Such changes to B do not reveal anything new to an adversary, who can anyway estimate aggregate load.

Supporting writes. PANCAKE handles updates (writes) to keys in KV by borrowing a standard technique from the ORAM literature [23]: treat each access as a read followed by a write. After the client receives the B encrypted values from the server corresponding to the batch, it decrypts, possibly updates, then re-encrypts the values and sends them back to the server. Each access therefore consists of a fixed-size batch of reads followed by a fixed-size batch of writes to the same labels. When a key has multiple replicas and its value is updated, the client adds it to the UpdateCache to track which of its replicas still need to be updated (updating all replicas at once leaks information). PANCAKE consults the UpdateCache every time it does a writeback to ensure all updates propagate. Once all of a key's replicas have been updated, its entry is removed from the cache. Note that PANCAKE can use any access (fake or real) to opportunistically propagate updates.

4.3 Security Analysis

Intuitively, PANCAKE security stems from the following three points. (1) The cryptographic security of F as a pseudorandom function and E as a (randomized) authenticated encryption scheme. This ensures that the keys $F(k, j)$ appear random and that nothing leaks about values. (2) Assuming client requests are distributed according to π and that our estimate $\hat{\pi}$ of π is sufficiently good, each individual access is uniformly distributed over KV$'$ by Equation 1. (3) Fake and real queries cannot be distinguished by the server (i.e., none of the coin tosses q_{type} can be inferred). The third point requires that the number and timing of accesses observed by the server be independent of the coin tosses. We do not attempt to hide the time at which an access is made by a client, but the timing should be independent of which key a client requests and which accesses are fake or real — thus, similar to ORAM designs [9], PANCAKE implementations must be constant-time.

Formal analysis. To provide a formal analysis, we introduce a security definition called real-versus-random indistinguishability under chosen distribution attack or ROR-CDA. A formal game-based definition of ROR-CDA is given in Appendix A. Briefly, in the real world the adversary is given PANCAKE's encryption of the KV store KV$'$ and a transcript τ generated by running Batch on q samples from π (where Batch uses $\hat{\pi}$). In the ideal world, the adversary is given a database consisting of random bit strings and a transcript of $q \cdot B$ uniformly random accesses.

Achieving this security goal rules out attacks based on access pattern leakage. Take frequency analysis as an example. If ROR-CDA holds, the frequency with which any label is accessed is independent of the label itself. Thus, frequency

analysis and any other attacks which rely on computing the most likely access will fail — all accesses are equally likely, so it is impossible to do better than baseline guessing.

The following theorem establishes the ROR-CDA security of PANCAKE. The theorem reduces to the pseudorandom function security [22] of F, the real-versus-random indistinguishability [56] of E, and to the computational indistinguishability of π and $\hat{\pi}$.

Theorem 1 *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi, \hat{\pi}$ be distributions. For any q-query* ROR-CDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ such that*

$$\mathbf{Adv}^{ror\text{-}cda}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{prf}_{F}(\mathcal{B}) + \mathbf{Adv}^{ror}_{E}(\mathcal{C}) + \mathbf{Adv}^{dist}_{Q,\pi,\hat{\pi}}(\mathcal{D})$$

where F and E are the PRF and AE scheme used by PANCAKE*. Adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ each use Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Discussion. Details of our formal analysis, including the proof of Theorem 1, are presented in Appendix A. Here we make some salient observations.

Our theorem is "parameterized" by q, π, $\hat{\pi}$. It applies to any distribution π, and provides security up to the ability to accurately estimate it. In the best case, estimation is perfect, $\hat{\pi} = \pi$, and Theorem 1 is optimal in the sense that the only way to break PANCAKE is to break one of the underlying cryptographic tools. Even if our estimate is not perfect, it just needs to be good enough to be indistinguishable from the real distribution for a limited number of samples. While there exist distributions that are hard to estimate [7, 30, 59], real-world ones with heavy skew allow for sufficiently good estimation.

Our security model is highly pessimistic in that we assume the adversary has perfect knowledge of π. In reality they will not, and so we expect that in practice PANCAKE will provide even greater security than what our theory suggests.

4.4 Performance Analysis

PANCAKE incurs a bandwidth overhead of $B\times$, the size of each batch. With $\alpha = 1/n$, the server stores $2n$ replicas (including dummy replicas), so the server storage overhead is $2\times$. Note that PANCAKE bandwidth and server storage overheads are independent of the underlying data access distributions.

PANCAKE proxy storage and query latency overheads are related to query queue length, which itself is a function of batch size B. Experimentally, we observe a near-zero queue length for $B \geq 3$ (§6.3). This is supported by results in queuing theory: if we model the number of query arrivals per unit time as Poisson with mean λ, with $\delta = 1/2$ the number of departures per unit time with our scheme is also Poisson with mean $\lambda \cdot B/2$. Thus, our queue is well-modeled as M/M/1 with $\rho = \lambda/(\lambda \cdot B/2) = 2/B$. Applying standard results on steady-state behavior of such queues [16], as the number of queries goes to infinity, $\Pr[i \text{ queries in queue}] = (1 - \frac{2}{B})(\frac{2}{B})^i$. Thus the probability that a query waits for i queries ahead of it in the queue is exponentially vanishing in i.

The size of PANCAKE's UpdateCache depends on the query distribution, the threshold α, and the fraction of write queries. A loose bound on UpdateCache size is the number of keys with access probability greater than α. Intuitively, a pathological worst-case could occur when $n-1$ out of n keys have access probability slightly higher than $1/n$; in this case, each of the $n-1$ keys would have 2 replicas, and UpdateCache size could grow to $O(n)$ with very high write rates. We delegate a formal analysis of the worst-case UpdateCache size for specific distributions to future work, but note that our evaluation demonstrates that, for standard benchmark workloads comprising skewed distributions, the UpdateCache size turns out to be a small fraction ($< 5\%$) of the dataset size (§6.3).

5 Handling Dynamic Distributions

In the previous section, we showed how PANCAKE transforms any static distribution of key-value accesses into a uniformly-distributed one. For some applications, however, distributions will change over time. We now describe how PANCAKE adapts to changes in the query distribution. We start by describing the core dynamic adaptation technique in PANCAKE under the assumption that changes in distribution can be detected instantaneously (§5.1), prove PANCAKE security under this assumption (§5.2), and, finally, discuss some pragmatic issues of detecting changes in the underlying distribution (§5.3).

5.1 Adapting to Changes in Distribution

Once the new query distribution estimate $\hat{\pi}'$ is identified, PANCAKE must adapt to $\hat{\pi}'$ by smoothing it. We note that if all keys need the same number of replicas with $\hat{\pi}'$ as they need with $\hat{\pi}$, PANCAKE easily adapts to $\hat{\pi}'$ by recomputing the fake query distribution π_f as per Equation 1. However, when a key's probability $\hat{\pi}'(k)$ increases so much that $\hat{\pi}'(k) \geq R(k, \hat{\pi}, \alpha) \cdot \alpha$, then PANCAKE must change its number of replicas. Figure 3 shows an example for frequency smoothing of $\hat{\pi}$ and $\hat{\pi}'$; note that while key a gains a replica, the dummy key D loses one.

Adapting to changes in the query distribution while preserving both efficiency and security is challenging. One approach is downloading the entire database and re-running Init from Figure 2 with fresh keys. This is secure but prohibitively bandwidth-intensive, and queries cannot be serviced during reinitialization. One could instead act only on the replicas for keys whose probabilities have changed; this is insecure since accesses are non-uniform during the change. In Figure 3, if we only download a, add a new replica for it and delete one for D, then an adversary can infer that a grew in popularity.

Our solution builds on the latter approach, ensuring efficiency and security using an *online replica swapping* mechanism described next. To make replica swapping performant and secure, it must work in conjunction with two other techniques: adjusting the fake query distribution and caching

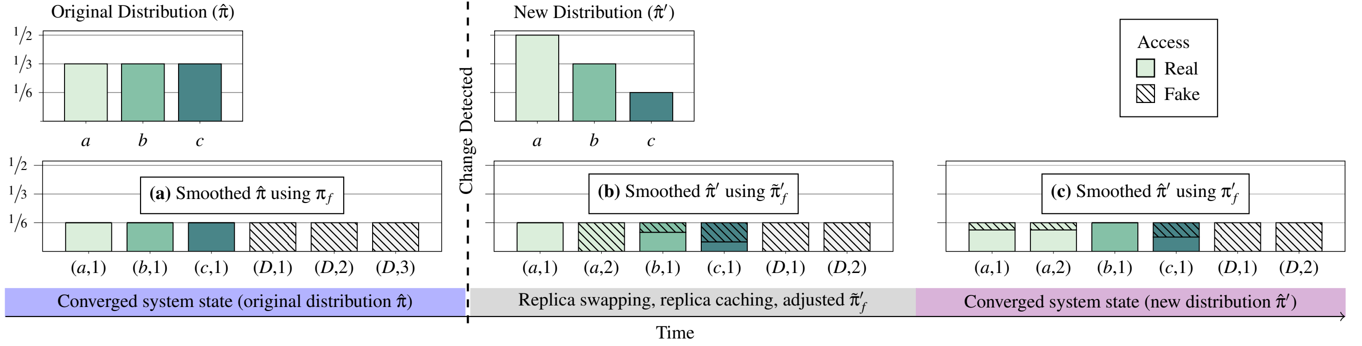


Figure 3: **Frequency smoothing for dynamic distributions.** (a) Smoothing for original distribution ($\hat{\pi}$) over replicas in KV' using fake distribution π_f. With $\hat{\pi}$, each of keys a,b,c has one replica, and the dummy key D has 3 replicas. (b) Detection of new distribution ($\hat{\pi}'$) over keys triggers replica swapping. During replica swapping, distribution over replicas in KV' is smoothed using an adjusted fake distribution $\tilde{\pi}'_f$: all real accesses to a are directed to $(a,1)$, and the real access probability is decreased until $(D,3)$ and $(a,2)$ are swapped. (c) Smoothing for new distribution ($\hat{\pi}'$) using new fake distribution π'_f, after replica swapping completes. Key a gains a replica, while D loses a replica.

replicas at the proxy.

Replica swapping. Our key insight in adapting to change in query distributions is that since the total number of replicas for any distribution is always exactly $2n$ (including dummy replicas), a transition from $\hat{\pi}$ to $\hat{\pi}'$ ensures that for each key k_i that gains a replica, there must be another key k_j that loses a replica. Therefore, handling changes in query distributions simply requires, for all such keys, reading k_j's value and writing it to one of k_j's replicas, a process we refer to as replica swapping. PANCAKE performs these swaps without revealing any information about the change by *piggybacking* the replica swaps atop normal client accesses. Maintaining uniform accesses during replica swapping requires changes to the Batch procedure and the fake access distribution as described in §4; we describe these changes next.

During replica swapping, the modified Batch uses two lists: G and L. G is the set of replicas that need to be created and L is the set of replicas that need to be removed. Formally, if S is the set of keys that must gain replicas, T is the set of keys that must lose replicas, and $R(k,\hat{\pi}',\alpha) = \lceil \hat{\pi}'(k)/\alpha \rceil$, then,

$$\mathsf{G} = \{(k,j)\,|\,k \in S,\ j \in [R(k,\hat{\pi},\alpha)+1,\ldots,R(k,\hat{\pi}',\alpha)]\}$$
$$\mathsf{L} = \{(k,j)\,|\,k \in T,\ j \in [R(k,\hat{\pi}',\alpha)+1,\ldots,R(k,\hat{\pi},\alpha)]\}$$

A pseudocode procedure for generating these lists from $\hat{\pi}$ and $\hat{\pi}'$ is given in the full version [25], along with a description of the modified Batch. It is not hard to see that $|\mathsf{G}| = |\mathsf{L}|$ always (since $|S| = |T|$), and that swapping each replica in L for one in G results in all keys having the right number of replicas under $\hat{\pi}'$. This swapping is done opportunistically by Batch: when a replica in L is read in a batch, either by a real or a fake query, its value is updated to the value associated with a replica in G during the writeback. For security reasons, PRF labels for replicas in G are not changed. Instead, PANCAKE maintains a mapping between the label of replicas in L and the replica in G it will be swapped with. On subsequent queries during the transition, Batch consults the mapping for the right labels. This metadata can be deleted after periodic rotation of the cryptographic keys. We describe key rotation in the full version [25]. When all swaps have occurred, we switch back to the normal Batch procedure for $\hat{\pi}'$.

As a concrete example of replica swapping, consider Figure 3. The set G contains the replica $(a,2)$, while L contains $(D,3)$. Note that both G and L could contain dummy replicas, depending on how the distribution changes. Batch would swap the replicas for keys a and D on the first access to $(D,3) \in \mathsf{L}$ by writing back an encryption of key a's value (because $(a,2) \in \mathsf{G}$) instead of a re-encryption of the dummy value D. To enable this, PANCAKE would maintain a mapping that indicates the label of $(a,2)$ is $F(D,3)$.

Adjustments to fake access distribution. Two more modifications are needed during the transition. First, we must use a different fake access distribution to ensure that reads to keys that have gained replicas always succeed. To see why this is necessary, consider again the example in Figure 3. If a query tries to read key a by accessing replica $(a,2)$ before the value of $(D,3)$ is changed, the read will return D's value instead of a's. Thus replica $(a,1)$ must be read, but forcing this makes $(a,1)$'s probability too high, violating security.

PANCAKE handles this by temporarily increasing the threshold α to $\alpha' = \max_k\{\pi'(k)/R(k,\hat{\pi},\alpha)\}$, and using a temporary fake access distribution $\tilde{\pi}'_f$ to satisfy Equation 1 with α'. For each $(k,j) \in \mathsf{G}$, we set $\tilde{\pi}'_f(k,j) = \frac{\alpha'}{2n\alpha'-1}$, and k's existing replicas have $\tilde{\pi}'_f = \frac{\alpha'-\hat{\pi}'(k)/R(k,\hat{\pi},\alpha)}{2n\alpha'-1}$. For other keys, $\tilde{\pi}'_f(k,j)$ is set to $\frac{\alpha'-\hat{\pi}'(k)/R(k,\hat{\pi}',\alpha)}{2n\alpha'-1}$.

Since $\alpha' \geq \alpha$, the real access probability $\delta = 1/2n\alpha'$ is lower during replica swapping. As such, this may lead to some real queries being delayed to later batches. This may

increase latency for some queries during replica swapping, but we show in §6.2 that replica swapping completes in a few minutes even for drastic changes in the distribution.

Replica caching. PANCAKE computes the mapping between each label in L and the replica in G it will be swapped with when the distribution change is detected. However, the actual values of replicas in G must be propagated to those in L during subsequent accesses to them. Without any additional mechanism, reads to keys with replicas in G may access incorrect values. To ensure correctness, when a replica in G is read during Batch, its value is cached at the proxy. This value is then propagated to the replica in L when it is next accessed, while the actual read is served from the cache.

Insertion and deletion of keys. We have assumed so far that the support size is fixed; interestingly, the replica swapping procedure can support changes in the set of keys. This can be viewed as a distribution change where $\text{supp}(\hat{\pi}') \neq \text{supp}(\hat{\pi})$. As long as PANCAKE is initialized with enough replicas to handle the maximum support size, new keys can be inserted by swapping a dummy replica for the new key, and vice versa for deletion. Some additional metadata is needed, but similar to the PRF label mapping it can be deleted as soon as cryptographic keys are rotated (details in the full version [25]).

5.2 Security Analysis

We prove that PANCAKE's accesses remain uniform even for time-varying distributions, under the assumption that changes in distributions can be detected instantaneously. We formalize our goal as a generalization of the static ROR-CDA security notion. We call this new notion "real-or-random security under chosen dynamic distribution attack", or ROR-CDDA. It is similar to its static analogue except that it uses two distributions π and π': after an adversarially chosen number of queries the distribution changes from π to π'. We let $\mathbf{Adv}^{\text{ror-cdda}}(\mathcal{A})$ be the ROR-CDDA advantage of an adversary $\mathcal{A}$. It captures the ability of $\mathcal{A}$ to distinguish between a real PANCAKE execution during a distribution change (ROR-CDDA1) and uniformly random accesses (ROR-CDDA0). The following theorem captures the ROR-CDDA security of PANCAKE.

Theorem 2 *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi, \pi', \hat{\pi}, \hat{\pi}'$ be distributions. For any q-query* ROR-CDDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}_1, \mathcal{D}_2$ such that*

$$\mathbf{Adv}^{\text{ror-cdda}}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{prf}_{F}(\mathcal{B}) + \mathbf{Adv}^{ror}_{E}(\mathcal{C}) + \mathbf{Adv}^{dist}_{Q,\pi,\hat{\pi}}(\mathcal{D}_1) + \mathbf{Adv}^{dist}_{Q,\pi',\hat{\pi}'}(\mathcal{D}_2)$$

where F and E are the PRF and AE scheme used by PANCAKE. *Adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}_1, \mathcal{D}_2$ each use at most Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Discussion. Full details of the definitions and a proof of Theorem 2 appear in Appendix A. We discuss here only one salient point regarding ROR-CDDA. ROR-CDDA models the shift from π to π' as happening and being detected instantaneously. This may not be realistic in some cases, even with state-of-the-art techniques in detecting distribution changes (as used in PANCAKE, discussed in next subsection). Thus, we cannot rule out the case where PANCAKE processes queries from π' before the change is detected (treating them like samples from $\hat{\pi}$). The distribution of these queries would be non-uniform, with bias related to the difference between π and π'. If the adversary knows the bias, using it in an attack would be possible but challenging—indeed, we are not aware of any published attacks that even consider the possibility of distribution changes.

5.3 Detecting Changes in Query Distribution

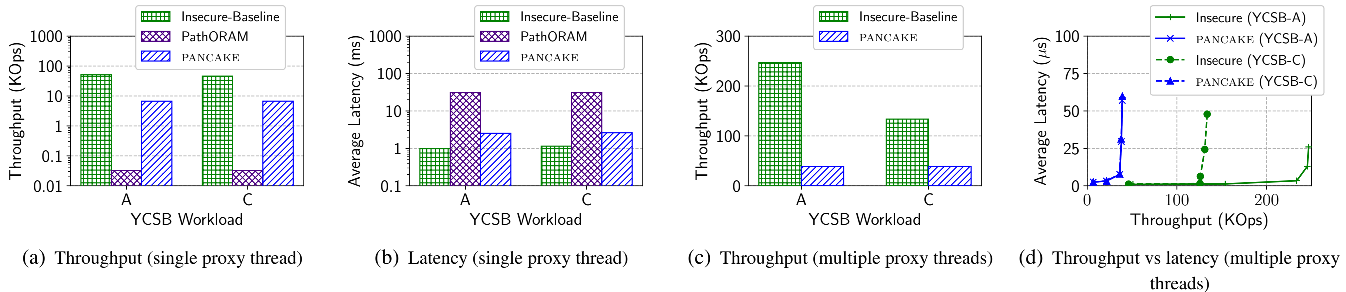
Detecting distribution changes using statistical tests is a well studied problem [34, 36, 61, 67]. While it is possible to have PANCAKE receive external signals (e.g., from an analyst) when the distribution changes, our implementation incorporates the two-sample Kolmogorov–Smirnov (KS) test [36, 61], a standard statistical tool, to detect such changes automatically. Specifically, recall that PANCAKE maintains a histogram H of observed accesses to maintain an estimate $\hat{\pi}$ for distribution π. In order to track changes to the distribution, PANCAKE additionally maintains a running histogram $H_{running}$ over a sliding window of the w latest accesses at the proxy. PANCAKE then uses KS test to determine when the underlying distribution corresponding to $H_{running}$ differs from $\hat{\pi}$. If the test indicates a change, PANCAKE uses the current $H_{running}$ snapshot to inform the estimate $\hat{\pi}'$ for the new distribution π'.

Detecting changes in distributions, and responding to these changes involves balancing security and efficiency. If the test is too sensitive the system will waste resources adjusting to spurious changes; on the other hand, as discussed above, an insensitive test could leak information about queries. While it is possible to use other statistical tests [67], or an ensemble of tests to navigate this tradeoff between performance and security, no statistical test is perfect. We present several evaluation results related to detecting and adapting to changes in query distribution, along with sensitivity analysis, in §6.2.

6 Evaluation

We now evaluate PANCAKE across a wide variety of scenarios, including main-memory and secondary storage-based data stores, static and dynamic distributions, deployment settings and workloads. We start by briefly describing the evaluation methodology, followed by detailed discussion of our results.

Compared approaches. We compare PANCAKE against two approaches: (1) an insecure baseline that provides no security guarantees, and (2) non-recursive PathORAM [63] (with $Z = 4$), a state-of-the-art ORAM. The former serves as an upper bound on PANCAKE performance, since it corresponds to a data store with no security overheads. The latter, on the



(a) Throughput (single proxy thread) (b) Latency (single proxy thread) (c) Throughput (multiple proxy threads) (d) Throughput vs latency (multiple proxy threads)

Figure 4: **Performance for in-memory server storage (Redis).** (a, b) PANCAKE's throughput is over 220× higher than PathORAM and within ∼6.8–7.6× of the insecure baseline for a single-threaded proxy; note that the y-axis is in log-scale. (c, d) With multiple proxy threads, PANCAKE's peak throughput is within 3.4–6.3× and latency within 2.3–2.6× of the insecure baseline.

other hand, is the state-of-the-art design that provides security under our model (as well as under stronger models where an adversary can actively inject its own queries). As discussed earlier, our comparison against the latter should be interpreted as highlighting the huge efficiency gap between countermeasures in the two threat models. We use batch size $B = 3$ for PANCAKE's Batch algorithm.

We compare these approaches using two representative storage backends: an in-memory KV store Redis [54], and a persistent SSD-based KV store RocksDB [55]. Our PathORAM deployment used an open-source implementation [14, 58]. For PathORAM and PANCAKE, client queries are forwarded to the data store via a proxy server; for the insecure baseline, client queries are forwarded to the backend storage server without any intermediary proxy.

The PathORAM implementation used in our evaluation [14, 58] is single-threaded. TaoStore [57] and ConcurORAM [13] implement multi-threaded PathORAM; we omit results for them since they employ specialized storage backends adapted for ORAMs, eschewing fair comparison with backends we investigate. We note, however, that the performance reported in [13, 57] is at least an order of magnitude lower than PANCAKE even with specialized storage backends.

Experimental setup. Our experiments run on Amazon EC2. The storage backend runs on a single t3.2xlarge instance with 8 vCPUs, 32GB RAM, and 1Gbps network and disk bandwidth. We use 1Gbps links and proxy/client machines with sufficient resources (r4.8xlarge instances with 32 vCPUs, 244GB RAM, 10Gbps network bandwidth) to highlight the impact of network bandwidth as a bottleneck.

Dataset and workloads. We use the Yahoo! Cloud Serving Benchmark (YCSB) [17], a standard benchmark for KV stores, to generate the datasets and workloads. The dataset contains 2^{20} KV pairs, with 8B keys and 1KB values. We confine our dataset size to 1GB since PathORAM has prohibitively large initialization times ($>$ 24 hours) and storage overheads ($> 10\times$) with larger datasets, while PANCAKE performance is essentially independent of dataset size.

We evaluate system throughput and latency using two YCSB workloads: Workload A (50% reads, 50% writes) and C (100% reads). These workloads represent two extremes in read-write proportions; other YCSB workloads either have intermediate read-write proportions (e.g., Workload B, D) or contain queries not supported by PANCAKE (e.g., Workload E). YCSB uses a Zipf distribution over key accesses (with skewness parameter $= 0.99$, i.e., very skewed), which is representative of access patterns in real-world deployments [17].

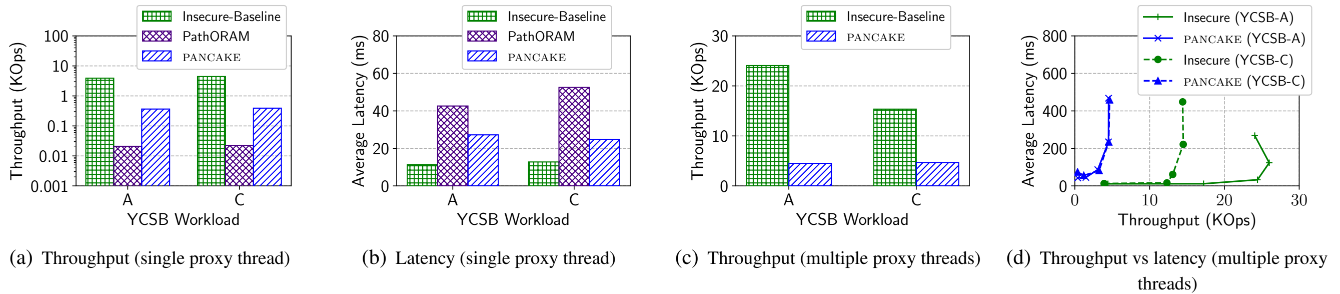
6.1 Performance for Static Distributions

We first compare the performance for different approaches with various storage backends under static query distributions.

In-memory server storage (Redis, Figure 4). With a single proxy thread, PANCAKE and PathORAM performance is bottlenecked by the proxy. For this evaluation setting, PathORAM achieves throughput ∼1600× lower compared to the insecure baseline. This is because PathORAM issues 160 storage backend requests ($= Z \log_2 N$, $Z = 4$, $N = 2^{20}$) for every client request, along with complex tree and stash management.

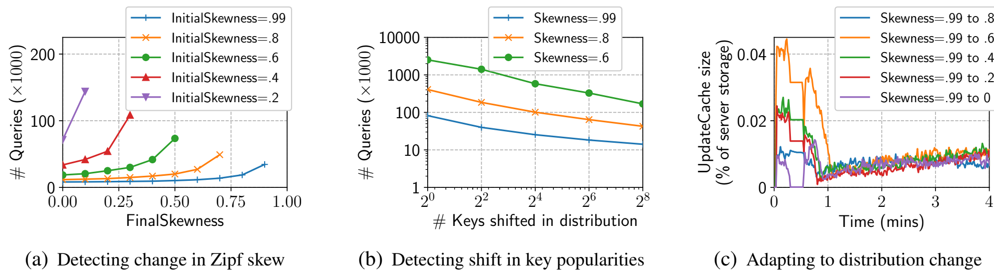
PANCAKE achieves significantly better throughput (as much as 229× better) compared to PathORAM. In comparison to the insecure baseline, PANCAKE average latency is within 2.3–2.6× and throughput is within 6.8–7.6× (Figure 4(a), 4(b)). This is a cumulative effect of three factors: (1) 3× bandwidth overhead due to batch size $B = 3$, (2) 2× overhead since each request generates a read and a write request in PANCAKE, and (3) overheads due to encryption/decryption. Our evaluation confirms that adding encryption/decryption to the insecure baseline brings PANCAKE's relative throughput overhead to 6×. We note that PANCAKE's 99th percentile latency (not shown in graphs) is relatively higher (within 4.1–5.6× the insecure baseline) due to queueing delays from PANCAKE's Batch algorithm. We note that if reducing tail latency were the goal, one can achieve that at the cost of higher bandwidth overheads by increasing B (§6.3).

With multiple proxy threads, PANCAKE peak throughput is within 3.4× of baseline for the read-only workload (YCSB Workload C) — a factor of 2 better than the single proxy thread. This reduction in relative overhead is due to the shift in performance bottleneck to the network bandwidth in the multi-threaded setting. We note that all network links are *full-*



(a) Throughput (single proxy thread) (b) Latency (single proxy thread) (c) Throughput (multiple proxy threads) (d) Throughput vs latency (multiple proxy threads)

Figure 5: **Performance for SSD-based server storage (RocksDB).** (a, b) PANCAKE's throughput is 17.3× higher than PathORAM and within ∼10.7–11.3× of the insecure baseline for a single-threaded proxy; note that the y-axis is in log scale for (a). (c, d) Using multiple proxy threads, PANCAKE's peak throughput is within 3.3–5.3× and average latency within 2–2.4× of the insecure baseline.



(a) Detecting change in Zipf skew (b) Detecting shift in key popularities (c) Adapting to distribution change

Figure 6: **Handling dynamic distributions.** (a, b) PANCAKE detects larger distribution changes in fewer queries, relative to smaller changes. (c) PANCAKE can adapt from a skewed to uniform distribution with UpdateCache size $<$ .05% of server storage over evaluated workloads.

duplex. As such, although every read request generates a read and a write request in PANCAKE, write requests saturate the network bandwidth *to* the server, while read responses saturate the bandwidth *from* the server, i.e., reads and writes saturate different directions of the link. In contrast, the read-only workload for the insecure baseline is only able to saturate one direction of the link. For the 50% read, 50% write workload (YCSB Workload A), PANCAKE's throughput remains the same, while baseline throughput increases by ∼1.8×, since the baseline can now also exploit full-duplex links. The throughput versus latency variation (Figure 4(d)) shows that the throughput reported in Figure 4(c) corresponds to the knee point in the curve (i.e., the sweet spot for latency and throughput) for both the insecure baseline and PANCAKE.

SSD-based server storage (RocksDB, Figure 5). With RocksDB, PathORAM achieves throughput ∼196× worse than the insecure baseline. Compared to the in-memory case, the slight improvement relative to the insecure baseline is due to PathORAM overheads overlapping with higher overheads of accessing data off SSD. As such, the proxy overheads account for a smaller fraction of the end-to-end performance. PANCAKE's performance is ∼17.3× better than PathORAM and within ∼11.3× of the baseline.

With multiple proxy threads, PANCAKE peak throughput is within 3.3× of the insecure baseline for read-only workload and within 5.3× for the 50% read, 50% write workload similar to the in-memory case. Figure 5(d) confirms that throughput in Figure 5(c) corresponds to the performance knee-point.

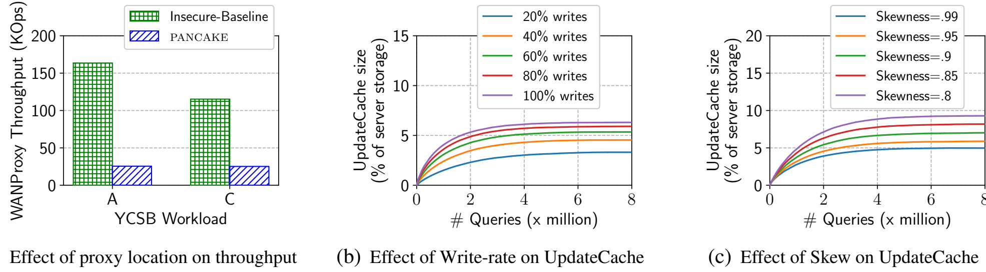
Storage overheads. PANCAKE's server storage requirements are ∼4× lower than the non-recursive PathORAM implementation that we use ($= 2 \cdot Z \cdot N$, for $Z = 4$) and within 2× of the insecure baseline, consistent with the theoretical storage overheads for both approaches. PANCAKE's proxy storage requirement is a small fraction of the total storage footprint (∼1%), similar to PathORAM (∼0.33%) for all evaluated workloads. PANCAKE proxy storage overheads due to the UpdateCache is dependent on write-rates and skew in the distribution; we evaluate these in §6.3.

6.2 Adapting to Dynamic Distributions

We evaluate PANCAKE's ability to detect and adapt to changes in distribution in isolation (i.e., without the effect of writes) using YCSB Workload-C (read-only). We present results for the in-memory storage backend. We set the sliding window size w for the running histogram $H_{running}$ to 10 million queries, and the confidence interval for the KS test to 95%.

Detecting distribution change. We quantify the cost of detecting distribution change in terms of the number of queries that must be observed before the change is detected. Figure 6(a) measures this as the skewness for the Zipf distribution is varied; as expected, the test detects *larger* changes in distribution (e.g., skewness drop from 0.99 to 0.0) in *fewer* queries, relative to much smaller changes (e.g., skewness drop from 0.99 to 0.9). This is consistent with the KS test's sensitivity.

In Figure 6(b), we shift the Zipf key popularities by κ, i.e., the most popular key becomes the κth most popular key, the second most popular key becomes the $(\kappa+1)^{th}$ most popular, and so on, while the κ least popular keys become the most popular keys. This models changes in real-world access



(a) Effect of proxy location on throughput (b) Effect of Write-rate on UpdateCache (c) Effect of Skew on UpdateCache

Figure 7: (a) Due to asymmetric and unpredictable available download and upload speeds over the Internet, both the insecure baseline and PANCAKE observe reduced throughput (0.65–0.85×) for the WAN setting when compared to the cloud setting. (b, c) UpdateCache size increases with write rate for a fixed Zipf distribution (skewness = 0.99) and decreases as skew increases for a fixed write-rate (50%), but remains well below 10% of server storage for all evaluated workloads.

patterns where some items can suddenly become more popular [4]. Again, we observe that larger changes in distribution (e.g., shift by $\kappa = 256$) is detected in fewer queries (e.g., a few hundred thousand) than smaller changes (e.g., shift by $\kappa = 1$, which may require millions of queries).

The results for both settings show that PANCAKE's mechanism for detecting changes in distribution works well in practice, e.g., at 100K queries per second, PANCAKE can detect changes in 1–2 seconds.

Adapting to distribution change. We evaluate PANCAKE overheads in adapting to dynamic distributions when the underlying distribution changes. In particular, we change the distribution from high skewed (skewness parameter = 0.99) to smaller skews, with the extreme case of a pure uniform access pattern (skewness parameter = 0).

Our results show that PANCAKE can adapt to even drastic changes in distribution (Zipf to pure uniform) in less than $\sim$ 25 minutes (for updating newly assigned replicas across all keys), while using $< 0.05\%$ of the server storage at the proxy (Figure 6(c)). This is interesting for two reasons: (1) PANCAKE observes only a negligible increase in proxy storage during the adaptation period, and (2) the adaptation occurs in the background, i.e., without stopping query execution, and in fact piggybacks on the query execution to carry out the adaptation. As such, higher query rates would lead to even faster adaptation to changes in distribution.

6.3 Performance Sensitivity to Parameters

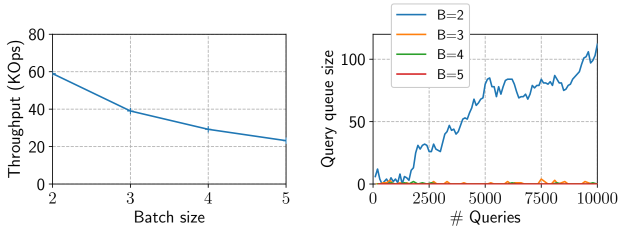
We now analyze the sensitivity of PANCAKE's performance and storage overheads to various parameters. We highlight differences in our experimental setup wherever necessary.

Effect of proxy location (Figure 7(a)). We measure the impact of proxy location relative to the storage server by placing the proxy in a university network, connected to the cloud storage via the Internet. The proxy server has a 16-core 2.60GHz Intel Xeon CPU, 128GB RAM and 1Gbps access link to the Internet. Figure 7(a) measures the throughput for this setup (which we refer to as WAN proxy) using multiple proxy threads. The throughput for WAN-Proxy is slightly lower than Cloud-Proxy (Figure 4(c)), since the available bandwidth over the Internet was lower than 1Gbps and often unstable. Moreover, the measured upload bandwidth was lower than the download bandwidth over the Internet, which resulted in slightly lower throughput ($\sim 0.65\times$) for PANCAKE, and the insecure baseline for Workload A (50% reads, 50% writes).

Impact of write rates (Figure 7(b)) and request distributions (Figure 7(c)) on UpdateCache. We quantify the proxy storage overhead due to UpdateCache by measuring its size for varying fractions of write rates and for varying skew in underlying distribution across keys. Figure 7(b) shows that UpdateCache size is well below 10% of the server storage even with 100% writes. For the more realistic case of lower write rates, the storage overhead is much lower, e.g., with 20% write rate, the overhead reduces to $< 3\%$ of server storage.

While most real-world distributions are heavily skewed [4], the fraction of keys with > 1 replica in PANCAKE increases with decrease in skew. This can lead to an increase in UpdateCache size, since PANCAKE caches values for such keys while propagating writes to their replicas. We evaluate this overhead by measuring UpdateCache size for workloads with different degrees of skew for the YCSB Workload A (50% read-50% write). Figure 7(c) shows that decreasing skewness from 0.99 to 0.8 increases UpdateCache size from 5% to 9% of server storage, i.e., UpdateCache size remains a small fraction of server storage even at low skew.

Effect of batch-size B (Figure 8(a)-8(b)). Recall from §4.4 that, for a batch size of B, PANCAKE incurs bandwidth overhead of $B\times$; Figure 8(a) shows that when network bandwidth is the bottleneck, PANCAKE throughput degrades proportionally to the value of B. At the same time, larger B values leads to lower tail latency, since requests wait in the query queue for fewer batches — while $B = 2$ leads to an unstable queuing system (Figure 8(b)), $B > 2$ observes little or no queuing delays. B thus exposes a tradeoff between tail latency and throughput, where $B = 3$ provides a sweet spot for both. We do not evaluate latency vs. batch size since latency is tied to query inter-arrival times. For fixed inter-arrival times, latency overheads can be extrapolated from Figure 8(b).



(a) Effect of batch size on throughput (b) Effect of batch size on query queue

Figure 8: (a, b) Impact of batch size on PANCAKE throughput and query queue size. See §6.3 for discussion.

7 Discussion

PANCAKE is a first step toward designing high-performance data stores that are secure against access pattern attacks by passive persistent adversaries. In this section, we discuss several possible avenues for future research.

Correlated accesses. Our security analysis for PANCAKE relies on the assumption that queries are independent; in some application contexts, queries can be correlated. To the best of our knowledge, frequency analysis for correlated queries has not been explored. We present some preliminary results in the full version [25]; specifically, we show that security in a variant of ROR-CDA that allows arbitrary correlations is equivalent to ORAM security, and must therefore suffer from the same lower bounds on ORAM efficiency. However, this result relies on the adversary being able to construct very specific and artificial query correlations. We believe that we need new technical tools to explore access patterns attacks under realistic query correlations.

Stronger adversaries. PANCAKE targets a security model where the attacker does not tamper with data or do rollback attacks. PANCAKE's use of authenticated encryption means tampering is detectable, and preventing rollbacks is possible via authenticated operation counters. However, unlike ORAM, PANCAKE does not provide security against adversaries that can inject their own queries [12, 68]. We discuss how such chosen-query attacks could work on PANCAKE, and how it mitigates such attacks to some extent in the full version [25]. Informally, we show that PANCAKE does no worse than other efficient schemes against such attacks.

Dynamic distributions. For the case of dynamic distribution, PANCAKE's security is proven under the assumption that changes in distribution happen instantaneously and can be detected instantaneously. While our evaluation suggests that PANCAKE can detect changes in distribution within a few seconds, it would be nice to generalize our analysis to capture more gradual changes in distribution.

Improved proxy implementation. The current PANCAKE implementation uses a stateful proxy that stores distributions $(\hat{\pi}, \pi_f)$, key→replica counts, and pending writes in the UpdateCache. It would be interesting to explore implementations that allow the proxy to be more scalable (e.g., using a distributed proxy implementation) and fault tolerant (e.g., using techniques similar to [18]).

Variable-sized values. Similar to existing ORAM designs, to avoid attacks based on length leakage, current PANCAKE design assumes that values stored in the data store are fixed-size or have been padded to a fixed maximum length. While this is useful for many applications (e.g., values have fixed size when storing tweets, and storage systems like DynamoDB have upper bounds on value sizes), forcing values to be padded can cause prohibitive space overheads if there is a large difference between the largest and smallest values. It would be interesting to extend PANCAKE design to avoid storage overheads while protecting against attacks based on length leakage.

Hiding access patterns in cache-based systems. Many real-world systems execute queries on SSD-based storage with in-memory cache (e.g., MySQL server with memcached as a cache [44]). The problem of hiding access pattern seems to be at odds with achieving high performance in such deployment settings — intuitively, for workloads with skewed access patterns, it is possible to achieve performance gains by serving popular keys from the faster cache [69] at the cost of leaking that keys in cache are accessed more frequently than those not in cache. Hiding access patterns requires all keys to be accessed uniformly thus invalidating the benefits of a cache without any additional mechanism. Our preliminary evaluation, presented in the full version [25], shows that depending on the distribution and available cache size, existing systems including PANCAKE can observe as much as an order-of-magnitude throughput degradation compared to the insecure baseline that can effectively exploit the benefits of cache. It would be interesting to explore techniques that avoid such performance degradation while providing security against access pattern attacks.

8 Conclusion

In this paper, we explored a novel frequency-smoothing based countermeasure against access pattern attacks on outsourced storage in a new formal security model. We instantiated this approach in a new system called PANCAKE, the first to resist access pattern attacks by persistent passive adversaries while maintaining low constant factor overheads in storage and bandwidth. As such, PANCAKE's throughput is 229× higher than PathORAM, and within 3–6× of insecure baselines.

Acknowledgements

We thank the Usenix Security reviewers for their insightful feedback. We also thank our shepherd Amir Rahmati for his help with revisions to the paper. We thank Haris Mughees for his help in the early stages of the project. Grubbs was supported by NSF DGE-1650441. This work was in part supported by NSF grants 1704742, 1704296, 1514163, a Google Faculty Research Award, and a gift from Snowflake.

References

[1] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling Queries on Compressed Data. In *NSDI*, 2015.

[2] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *EuroSys*, 2011.

[3] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious ram. In *EUROCRYPT*, 2020.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *SIGMETRICS*, 2012.

[5] Baffle. https://baffle.io.

[6] Mihir Bellare, Anand Desai, Eron Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *FOCS*, 1997.

[7] Gyora M. Benedek and Alon Itai. Learnability with respect to fixed distributions. *Theor. Comput. Sci.*, 1991.

[8] Vincent Bindschaedler, Paul Grubbs, David Cash, Thomas Ristenpart, and Vitaly Shmatikov. The tao of inference in privacy-protected databases. IACR ePrint, 2017. http://eprint.iacr.org/2017/1078.

[9] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In *CCS*, 2015.

[10] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *ITCS*, 2016.

[11] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, et al. TAO: Facebook's Distributed Data Store for the Social Graph. In *ATC*, 2013.

[12] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In *CCS*, 2015.

[13] Anrin Chakraborti and Radu Sion. Concuroram: High-throughput stateless parallel multi-client oram. In *NDSS*, 2019.

[14] Zhao Chang, Dong Xie, and Feifei Li. Oblivious ram: a dissection and experimental evaluation. *VLDB*, 2016.

[15] Ciphercloud. http://www.ciphercloud.com/.

[16] Jacob Willem Cohen and Anthony Browne. *The single server queue*. 1982.

[17] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*, 2010.

[18] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *OSDI*, 2018.

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *SOSP*, 2007.

[20] Deep Learning Meets Heterogeneous Computing. https://bit.ly/3hCoPz8.

[21] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *OSDI*, 2016.

[22] Oded Goldreich, Shaffi Goldwasser, and Silvio Micali. How to construct random functions. *JACM*, 1986.

[23] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *JACM*, 1996.

[24] How Google Search works. https://bit.ly/3hGWt70.

[25] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. Technical report, 2020. https://github.com/pancake-security.

[26] Paul Grubbs, Marie-Sarah Lacharité, Brice Minaud, and Kenneth G Paterson. Learning to reconstruct: Statistical learning theory and encrypted database attacks. In *IEEE S&P*, 2019.

[27] Paul Grubbs, Thomas Ristenpart, and Vitaly Shmatikov. Why your encrypted database is not secure. In *HotOS*, 2017.

[28] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. TCP≈RDMA: Cpu-efficient remote storage access with i10. In *NSDI*, 2020.

[29] MS Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: ramification, attack and mitigation. In *NDSS*, 2012.

[30] Michael Kearns, Yishay Mansour, Dana Ron, Ronitt Rubinfeld, Robert E. Schapire, and Linda Sellie. On the learnability of discrete distributions. In *STOC*, 1994.

[31] Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O'Neill. Generic attacks on secure outsourced databases. In *CCS*, 2016.

[32] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Blowfish: Dynamic storage-performance tradeoff in data stores. In *NSDI*, 2016.

[33] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. Zipg: A memory-efficient graph store for interactive queries. In *SIGMOD*, 2017.

[34] Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting change in data streams. In *VLDB*, 2004.

[35] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash $\approx$ local flash. *SIGARCH*, 2017.

[36] Andrey Kolmogorov. Sulla determinazione empirica di una lgge di distribuzione. *Inst. Ital. Attuari, Giorn.*, 1933.

[37] Evgenios M Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. Data recovery on encrypted databases with k-nearest neighbor query leakage. In *IEEE S&P*, 2019.

[38] Marie-Sarah Lacharité and Kenneth G. Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically-encrypted data. IACR ePrint, 2017. https://eprint.iacr.org/2017/1068.

[39] Kasper Green Larsen, Tal Malkin, Omri Weinstein, and Kevin Yeo. Lower bounds for oblivious near-neighbor search. *arXiv preprint arXiv:1904.04828*, 2019.

[40] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO*. Springer International Publishing, 2018.

[41] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *EuroSys*, 2014.

[42] Charalampos Mavroforakis, Nathan Chenette, Adam O'Neill, George Kollios, and Ran Canetti. Modular order-preserving encryption, revisited. In *SIGMOD*, 2015.

[43] MongoDB. http://www.mongodb.org.

[44] InnoDB memcached Plugin. https://bit.ly/3edTmRD.

[45] Navajo Systems. http://tinyurl.com/y85obds6.

[46] Muhammad Naveed, Seny Kamara, and Charles V Wright. Inference attacks on property-preserving encrypted databases. In *CCS*, 2015.

[47] Neo4j. http://neo4j.com/.

[48] Antonis Papadimitriou, Ranjita Bhagwan, Nishanth Chandran, Ramachandran Ramjee, Andreas Haeberlen, Harmeet Singh, Abhishek Modi, and Saikrishna Badrinarayanan. Big data analytics over encrypted datasets with Seabed. In *OSDI*, 2016.

[49] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. What storage access privacy is achievable with small overhead? In *PODS*, 2019.

[50] Giuseppe Persiano and Kevin Yeo. Lower bounds for differentially private rams. In *EUROCRYPT 2019*, 2019.

[51] Perspecsys: A Blue Coat Company. http://perspecsys.com/.

[52] Rishabh Poddar, Tobias Boelter, and Raluca Ada Popa. Arx: an encrypted database using semantically secure encryption. *VLDB*, 2019.

[53] Raluca Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

[54] Redis. http://www.redis.io.

[55] RocksDB. http://rocksdb.org.

[56] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In *EUROCRYPT*, 2006.

[57] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. Taostore: Overcoming asynchronicity in oblivious data storage. In *IEEE S&P*, 2016.

[58] A unified testbed for evaluating different Oblivious RAM. https://github.com/InitialDLab/SEAL-ORAM.

[59] Rocco A Servedio. Lower bounds for learning discrete distributions.

[60] Skyhigh Networks. https://www.skyhighnetworks.com/.

[61] Nikolai V Smirnov. Estimate of deviation between empirical distribution functions in two independent samples. *Bulletin Moscow University*, 1939.

[62] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE S&P*, 2013.

[63] Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious ram protocol. In *CCS*, 2013.

[64] The Infrastructure Behind Twitter: Scale. https://bit.ly/2zLrDsI.

[65] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. Building an elastic query engine on disaggregated storage. In *NSDI*, 2020.

[66] Mor Weiss and Daniel Wichs. Is there an oblivious ram lower bound for online reads? In *TCC*, 2018.

[67] Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1945.

[68] Yupeng Zhang, Jonathan Katz, and Babis Papamanthou. All your queries are belong to us: the power of file injection attacks. In *USENIX Security*, 2016.

[69] Wenting Zheng, Frank Li, Raluca Ada Popa, Ion Stoica, and Rachit Agarwal. MiniCrypt: Reconciling encryption and compression for big data stores. In *EuroSys*, 2017.

A Security Proofs

In this appendix, we give some technical preliminaries and then prove Theorems 1 and 2.

Technical preliminaries. Throughout, we will use the concrete security approach [6]. For a (keyed) function $F: \mathcal{K} \times \{0,1\}^* \to \{0,1\}^m$ and adversary $\mathcal{A}$, we define the *pseudo-random function* (PRF) advantage relative to two games. In game PRF1, $\mathcal{A}$ has access to an oracle that accepts inputs from $\{0,1\}^*$ and outputs the PRF value on that point and a uniformly random key (which is the same for all queries). In game PRF0, $\mathcal{A}$'s oracle is a (lazy-sampled) random function from $\{0,1\}^*$ to $\{0,1\}^m$. We define $\mathcal{A}$'s PRF advantage to be

$$\mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{A}) = \left|\Pr\left[\mathrm{PRF1}^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathrm{PRF0}^{\mathcal{A}} \Rightarrow 1\right]\right|$$

where the probability is taken over the random choice of key (in PRF1) or lazy-sampled random function (in PRF0) and the adversary's internal random coins. Below, we will leave implicit the coin spaces involved in probabilities.

An *authenticated encryption with associated data* (AEAD) scheme $E = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ is a triple of algorithms. The function $E.\mathsf{KeyGen}$ takes no inputs and outputs elements of $\mathcal{K}$. The function $E.\mathsf{Enc}$ takes a key from $\mathcal{K}$, a plaintext from $\mathcal{M}$, (optionally) some associated data from $\mathcal{A}$, and outputs ciphertexts in $\mathcal{C}$. The function $E.\mathsf{Dec}$ takes a key from $\mathcal{K}$, a ciphertext from $\mathcal{C}$, (optionally) some associated data from $\mathcal{A}$, and outputs a plaintext in $\mathcal{M}$ or $\perp$.

We additionally require AEAD schemes to have a function len which takes a positive integer ℓ representing a plaintext length and outputs the length of any ciphertext of a plaintext of length ℓ. Essentially, the length of any plaintext's ciphertext must be computable given *only* the plaintext length and nothing else. Most natural AEAD schemes have this property.

For an AEAD scheme E and adversary $\mathcal{A}$, we define the *real-or-random* (ROR) advantage of $\mathcal{A}$ against E relative to two games, ROR1 and ROR0. In the first $\mathcal{A}$ has access to an $E.\mathsf{Enc}$ oracle with uniformly random key, and in the second $\mathcal{A}$'s oracle returns uniformly random bit strings of length $\mathrm{len}(|m|)$ where $|m|$ is the length of the input. We define $\mathcal{A}$'s ROR advantage against E as

$$\mathbf{Adv}_E^{\mathrm{ror}}(\mathcal{A}) = \left|\Pr\left[\mathrm{ROR1}^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[\mathrm{ROR0}^{\mathcal{A}} \Rightarrow 1\right]\right| .$$

For a distribution π and adversary $\mathcal{D}$ that outputs a bit, let $\mathrm{DIST}_{q,\pi}^{\mathcal{D}}$ be the game that samples q times from π, runs $\mathcal{D}$ on the resulting outputs, and outputs $\mathcal{D}$'s output. For two distributions π, π' with $\mathrm{supp}(\pi) = \mathrm{supp}(\pi')$, we measure their q-sample indistinguishability from an adversary $\mathcal{D}$ via the advantage measure

$$\mathbf{Adv}_{q,\pi,\pi'}^{\mathrm{dist}}(\mathcal{D}) = \left|\Pr\left[\mathrm{DIST}_{q,\pi}^{\mathcal{D}} \Rightarrow 1\right] - \Pr\left[\mathrm{DIST}_{q,\pi'}^{\mathcal{D}}\right]\right| .$$

This just captures the computational indistinguishability of the two distributions, given q samples from them.

Frequency smoothing KV schemes. Recall from §4 that PANCAKE has two algorithms: Init and Batch. To model distribution estimation errors and adjustments made when distributions change (as per §5), we extend our formalism by introducing two further algorithms. More precisely, an encrypted KV scheme $\mathsf{EKV} = (\mathrm{Init}, \mathrm{Batch})$ is a pair of algorithms:

- A randomized initialization algorithm Init that takes as input an estimated distribution $\hat{\pi}$, a KV store KV, and a threshold α, and outputs an encrypted KV store KV', a fake distribution π_f, a function R, and a real query probability δ. We denote running this algorithm by $(\mathsf{KV}', \pi_f, R, \delta) \leftarrow\!\!\$\ \mathrm{Init}(\hat{\pi}, \mathsf{KV}, \alpha)$.
- A randomized, stateful batch query algorithm Batch that takes as input a key k, the function R that maps keys to replica counts, and outputs a batch of B labels. We denote running this algorithm by $(\ell_1, \ldots, \ell_B) \leftarrow\!\!\$\ \mathrm{Batch}(k)$. Note that to avoid notational clutter we omit from the notation the values $\hat{\pi}, \pi_f, \delta$ and the state that Batch relies upon.

We have assumed distributions have efficient representations, and abuse notation by using the same variables π, $\hat{\pi}$, π_f, etc., as both distributions and their representations. For any fixed distribution π, we assume that Init always outputs an encrypted KV store of a constant size n'. PANCAKE satisfies these assumptions; its algorithms were described in the body.

Notice that our formalization here only handles read queries. As discussed in the body, we perform writes by always doing write-backs. Thus, security analysis can be reduced to the read-only case, greatly simplifying our formalization and security definitions.

Security for static distributions. We now formalize our ROR-CDA definition for a fixed scheme $\mathsf{EKV} = (\mathrm{Init}, \mathrm{Batch})$.

```
ROR-CDA1^A_{q,π,π̂}:
KV ←$ A_1
(KV′, π_f, δ) ←$ Init(π̂, KV, α)
k_F ←$ K; k_AE ←$ K
For i in 1 to q:
    w_i ←$ π
    ℓ_1, ..., ℓ_B ←$ Batch(w_i)
    For j in 1 to B:
        τ_B[j] ← (ℓ_j, KV′[ℓ_j])
    τ[i] ← τ_B
b ←$ A_2(KV′, τ)
Return b
```

```
ROR-CDA0^A_q:
KV ←$ A_1
KV′ ← ∅
For i in 1 to n′:
    ℓ_i ←$ {0,1}^m
    v_i ←$ C
    KV′ ← KV′ ∪ {(ℓ_i, v_i)}
For i in 1 to q:
    For j in 1 to B:
        ℓ ←$ Labels(KV′)
        v ← KV′[ℓ]
        τ_B[j] ← (ℓ, v)
    τ[i] ← τ_B
b ←$ A_2(KV′, τ)
Return b
```

Figure 9: Security game for key value store schemes in the static distribution case. The threshold α is an implicit parameter of the left game. The procedures Init and Batch are as defined in Figure 2.

We measure the success of an adversary $\mathcal{A}$ in attacking EKV by its ability to distinguish between the games ROR-CDA1 and ROR-CDA0 as defined in Figure 9. The game ROR-CDA1 is parameterized by the number of queries q, the true distribution π and the estimated distribution $\hat{\pi}$. We also take α as an implicit parameter. The adversary runs and chooses a plaintext distribution, then Init is executed followed by a sequence of queries drawn according to π. A transcript of accesses is generated by Batch. The adversary runs again with input the encrypted database and transcript. The two adversary executions can share state.

In game ROR-CDA0, the adversary sees a randomly generated encrypted database and queries chosen uniformly at random. The advantage of $\mathcal{A}$ with q queries against EKV is defined as

$$\mathbf{Adv}^{\text{ror-cda}}_{\mathsf{EKV}}(\mathcal{A}) = \big|\Pr[\text{ROR-CDA1}^{\mathcal{A}}_q \Rightarrow 1] - \Pr[\text{ROR-CDA0}^{\mathcal{A}}_q \Rightarrow 1]\big| .$$

Next we state a key result, that PANCAKE achieves ROR-CDA security assuming estimation is sufficiently good. In particular this shows optimal security should $\hat{\pi} = \pi$.

Theorem 1 *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi, \hat{\pi}$ be distributions. For any q-query* ROR-CDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ such that*

$$\mathbf{Adv}^{\text{ror-cda}}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{prf}_{F}(\mathcal{B}) + \mathbf{Adv}^{ror}_{E}(\mathcal{C}) + \mathbf{Adv}^{dist}_{Q,\pi,\hat{\pi}}(\mathcal{D})$$

where F and E are the PRF and AE scheme used by PANCAKE*. Adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}$ each use Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Proof. We prove Theorem 1 using a series of standard cryptographic game transitions and reductions. We start with the game ROR-CDA1, replacing Init and Batch with the algorithms used in PANCAKE (see Figure 2). Game G_1 is the same as ROR-CDA1 except we replace the PRF F with a truly random function. The difference between the success of adversary $\mathcal{A}$ in these two games can be upper bounded by the advantage of a PRF adversary $\mathcal{B}$:

$$\left|\Pr\left[\text{ROR-CDA1}^{\mathcal{A}}_q \Rightarrow 1\right] - \Pr\left[G_1 \Rightarrow 1\right]\right| \leq \mathbf{Adv}^{\text{prf}}_{F}(\mathcal{B}) .$$

We then move to game G_2, which is the same as G_1 except we replace the authenticated encryption function E with a random function outputting strings in the ciphertext space. The difference between the success rate of $\mathcal{A}$ in G_2 and G_1 can be upper bounded by a real-or-random adversary $\mathcal{C}$ against the encryption scheme E:

$$\left|\Pr\left[G_1 \Rightarrow 1\right] - \Pr\left[G_2 \Rightarrow 1\right]\right| \leq \mathbf{Adv}^{\text{ror}}_{E}(\mathcal{C}) .$$

Finally we let G_3 be the same as G_2 except that we replace $\hat{\pi}$ with π everywhere. A straightforward reduction gives that

$$\left|\Pr\left[\text{ROR-CDA1}^{\mathcal{A}}_q \Rightarrow 1\right] - \Pr\left[G_1 \Rightarrow 1\right]\right| \leq \mathbf{Adv}^{\text{dist}}_{Q,\pi,\hat{\pi}}(\mathcal{D}) .$$

We now come to the core of the argument, that G_3 is identically distributed to ROR-CDA0. In G_3 all labels and values are random strings. Further, each of the accesses is a uniformly random choice from all possible labels.

To see this, observe that each access in a batch is independent and sampled from π with probability δ or π_f with probability $1-\delta$. By construction of the scheme as described in Equation 1, the probability of any replica being accessed is the same. Let $\hat{\tau}$ be a random variable representing the output of Batch on input a sample from π, and $\hat{\tau}_i$ be the ith access in the output. Then for all i and any replica (k, j)

$$\begin{aligned}\Pr[\hat{\tau}_i = (k,j)] &= \Pr[\hat{\tau}_i = (k,j) \mid q_{\text{type}} = 0] \cdot (1-\delta) \\ &\quad + \Pr[\hat{\tau}_i = (k,j) \mid q_{\text{type}} = 1] \cdot \delta \\ &= \frac{\alpha - \frac{\pi(k)}{R(k)}}{n'\alpha - 1} \cdot \frac{n'\alpha - 1}{n'\alpha} + \frac{\pi(k)}{R(k)} \cdot \frac{1}{n'\alpha} = \frac{1}{n'} .\end{aligned}$$

The theorem follows by the independence of the $\hat{\tau}_i$, and combining terms. ■

Security analysis for dynamic distributions. Next we analyze security for dynamic distributions. First we must extend the formalization of frequency-smoothing KV schemes from above to account for the extended semantics. Specifically the batch algorithm Batch can now take an optional additional input $\hat{\pi}'$, representing an updated distribution estimate. This signals to Batch that it must adjust to the new distribution. We denote running Batch as before when given this additional input by $\ell_1, \ldots, \ell_B \leftarrow\!\!\$\, \text{Batch}(\hat{\pi}', k)$. Recall that Batch is stateful and so when it gets a new estimate $\hat{\pi}'$, it also has access to the old estimate $\hat{\pi}$ as well as other state values. For PANCAKE, the Batch algorithm would use this information to run MakeReplicaLists and to setup its replica bookkeeping (refer to the full version for more detail). We now introduce

```
ROR-CDDA1^A_{q,π,π',π̂,π̂'}:
(KV, c) ←$ A_1
(KV', π_f, δ) ←$ Init(π̂, KV, α)
For i in 1 to c:
   w_i ←$ π
   ℓ_1, ..., ℓ_B ←$ Batch(w_i)
   For j in 1 to B:
      τ_B[j] ← (ℓ_j, KV'[ℓ_j])
   τ[i] ← τ_B
For i in c to q:
   w_i ←$ π'
   ℓ_1, ..., ℓ_B ←$ Batch(π̂', w_i)
   For j in 1 to B:
      τ_B[j] ← (ℓ_j, KV'[ℓ_j])
   τ[i] ← τ_B
b ←$ A_2(KV', τ)
Return b
```

```
ROR-CDDA0^A_q:
(KV, c) ←$ A_1
n ← supp(π)
KV' ← ∅ ; KV'' ← ∅
For i in 1 to n':
   ℓ_i ←$ {0,1}^m
   v_i ←$ C
   KV' ← KV' ∪ {(ℓ_i, v_i)}
For i in 1 to q:
   For j in 1 to B:
      τ_B[j] ←$ Labels(KV')
   τ[i] ← τ_B
b ←$ A_2(KV', τ)
Return b
```

Figure 10: Security games for dynamic key value store schemes. The threshold α is an implicit parameter of the left game.

a security definition ROR-CDDA or, real-or-random indistinguishability under chosen-dynamic-distribution attack. Game ROR-CDDA1 is now parameterized by the query number and four distributions $\pi, \hat{\pi}, \pi', \hat{\pi}'$. The adversary runs and can pick both a plaintext KV store and a change point $c \in [0, q]$. After the first c queries, keys switch from being sampled according to π to being sampled according to π' and Batch is run with the additional input $\hat{\pi}'$. The ROR-CDDA0 is the same as ROR-CDA0 except for the syntactic change that $\mathcal{A}_1$ outputs the additional value c. Otherwise the distribution over KV' (a KV store of uniform bit strings) and τ (qB uniform requests) are the same as before.

The ROR-CDDA advantage of an adversary $\mathcal{A}$ against a scheme EKV is defined as

$$\mathbf{Adv}^{\text{ror-cdda}}_{\mathsf{EKV}}(\mathcal{A}) = \Big|\Pr\left[\text{ROR-CDDA1}^{\mathcal{A}}_{q,\pi,\pi',\hat{\pi},\hat{\pi}'} \Rightarrow 1\right] - \Pr\left[\text{ROR-CDDA0}^{\mathcal{A}}_{q} \Rightarrow 1\right]\Big| .$$

One could easily extend this definition to handle a longer sequence of changes: our results extend to this setting as well. We note that the definition also implies that the transcript of queries is indistinguishable from one that is independent of the change point, meaning this information is hidden by schemes that meet the definition.

We now prove the following theorem about the dynamic version of PANCAKE.

Theorem 2 *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi, \pi', \hat{\pi}, \hat{\pi}'$ be distributions. For any q-query* ROR-CDDA *adversary $\mathcal{A}$ against* PANCAKE *we give adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}_1, \mathcal{D}_2$ such that*

$$\mathbf{Adv}^{\textit{ror-cdda}}_{\text{PANCAKE}}(\mathcal{A}) \leq \mathbf{Adv}^{\textit{prf}}_{F}(\mathcal{B}) + \mathbf{Adv}^{\textit{ror}}_{E}(\mathcal{C}) + \mathbf{Adv}^{\textit{dist}}_{Q,\pi,\hat{\pi}}(\mathcal{D}_1) + \mathbf{Adv}^{\textit{dist}}_{Q,\pi',\hat{\pi}'}(\mathcal{D}_2)$$

where F and E are the PRF and AE scheme used by PANCAKE*. Adversaries $\mathcal{B}, \mathcal{C}, \mathcal{D}_1, \mathcal{D}_2$ each use at most Q queries and run in time that of $\mathcal{A}$ plus a small overhead linear in Q.*

Proof. Similar to the proof of Theorem 1 above, we use game hops to replace the PRF labels and AE ciphertexts with random strings, and upper-bound the difference in advantage via the PRF and AE adversaries $\mathcal{B}$ and $\mathcal{C}$. We also replace $\hat{\pi}$ with π and $\hat{\pi}'$ with π' in game hops whose difference is upper bounded by the appropriate reductions to distinguishers $\mathcal{D}_1$ and $\mathcal{D}_2$. This brings us to a game G where labels and ciphertexts are random strings, but batches are generated using Batch with π on input samples from π (before the change) or with π' on input samples from π' (after the change).

We claim that the distribution of accesses in game G is uniformly random, the same as in ROR-CDDA0. Because the keys accessed in a batch are independent, it suffices to show a single access of a batch is uniform. Let $\hat{\tau}_i$ be the ith access of a batch generated by a query sampled from π'. Let (k, j) be any replica. Then to compute $\Pr[\hat{\tau}_i = (k, j)]$, there are a few cases. First recall that,

$$\Pr[\hat{\tau}_i = (k,j)] = \Pr[\hat{\tau}_i = (k,j) \mid q_{\text{type}} = 0] \Pr[q_{\text{type}} = 0] + \Pr[\hat{\tau}_i = (k,j) \mid q_{\text{type}} = 1] \Pr[q_{\text{type}} = 1]$$

where $q_{\text{type}} = 0$ means a fake query and $q_{\text{type}} = 1$ means a real query. There are three possible cases:

Case 1: k gained replicas and j is one of its existing replicas. Then the RHS above is equal to:

$$\frac{\alpha' - \pi'(k)/R(k)}{2n\alpha' - 1}\left(1 - \frac{1}{2n\alpha'}\right) + \frac{\pi'(k)}{R(k)} \cdot \frac{1}{2n\alpha'} = \frac{1}{2n}$$

Case 2: $(k, j) \in \mathsf{G}$. Then,

$$\Pr[\hat{\tau}_i = (k,j)] = \frac{\alpha'}{2n\alpha' - 1} \cdot \frac{2n\alpha' - 1}{2n\alpha'} = \frac{\alpha'}{2n\alpha'} = \frac{1}{2n} .$$

Case 3: (k, j) is either in L or is any other replica. In this case, $\Pr[\hat{\tau}_i = (k, j)] = 1/2n$ follows from Eq. 1. ■