ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL
ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

SHORTSTACK: Distributed, Fault-tolerant, Oblivious Data Access

Midhul Vuppalapati[*]
Cornell University

Kushal Babel[*]
Cornell University

Anurag Khandelwal
Yale University

Rachit Agarwal
Cornell University

Abstract

Many applications that benefit from data offload to cloud services operate on private data. A now-long line of work has shown that, even when data is offloaded in an encrypted form, an adversary can learn sensitive information by analyzing data access patterns. Existing techniques for oblivious data access—that protect against access pattern attacks—require a centralized and stateful trusted proxy to orchestrate data accesses from applications to cloud services. We show that, in failure-prone deployments, such a centralized and stateful proxy results in violation of oblivious data access security guarantees and/or in system unavailability. We thus initiate the study of distributed, fault-tolerant, oblivious data access.

We present SHORTSTACK, a distributed proxy architecture for oblivious data access in failure-prone deployments. SHORTSTACK achieves the classical obliviousness guarantee—access patterns observed by the adversary being independent of the input—even under a powerful passive persistent adversary that can force failure of arbitrary (bounded-sized) subset of proxy servers at arbitrary times. We also introduce a security model that enables studying oblivious data access with distributed, failure-prone, servers. We provide a formal proof that SHORTSTACK enables oblivious data access under this model, and show empirically that SHORTSTACK performance scales near-linearly with number of distributed proxy servers.

1 Introduction

Cloud services offer applications scalable, fault-tolerant, and easy-to-manage systems for storing and querying data. Many applications that benefit from offloading data to these cloud services operate on private data that can reveal sensitive information even when stored in an encrypted form [1–6]. An example is that of medical practices offloading patient health data to the cloud [7–9]—charts accessed by oncologists can reveal not only whether a patient has cancer, but also depending on the frequency of accesses (*e.g.*, the frequency of chemotherapy appointments), indicate the cancer's type and severity. Several such applications are subject to severe security concerns.

There is a large and active body of research on building systems for oblivious data access, that is, hiding not only the content of the data, but also data access patterns (*e.g.*, access frequency across data objects). These systems use one of the two techniques—Oblivious RAM [10–17] that enables oblivious data access against active adversaries but has bandwidth overheads that are logarithmic in the number of data objects, or Pancake [6, 18, 19] that enables oblivious data access against passive persistent adversaries with a small constant bandwidth overhead. Both of these techniques provide a powerful oblivious data access guarantee: an adversary observing all queries to and all responses from the cloud storage service observes uniform random accesses over the encrypted data objects. The challenge, however, is that both of these techniques require a centralized, *stateful*, proxy to orchestrate data access from applications to cloud services. Such a centralized and stateful proxy means that existing systems for oblivious data access suffer from two core issues (§3.1):

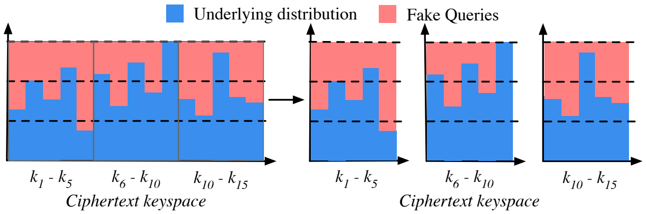
- *Security violation, or long periods of system unavailability during proxy failures:* The proxy being stateful means that, upon a failure, the proxy may lose state. We show in §3.1 that, if the proxy state is lost, naïvely restarting a new proxy and executing queries without restoring the state would lead to violation of oblivious data access security guarantees. To avoid such a security violation, upon restarting a new proxy, the state must be restored before executing any queries, *e.g.*, by downloading the entire data and metadata from the cloud, decrypting all the data, reconstructing the (ORAM or Pancake) data structure, re-encrypting all the data, and uploading all the data back to the storage service; this would lead to long periods of system unavailability.
- *Bandwidth and/or compute scalability bottlenecks:* Since the proxy receives multiple responses for each client query, it has bandwidth overheads ($\Omega(\log n)$ in ORAM [20–25] and 3× in Pancake [6]); and, since the proxy is responsible for both data encryption/decryption and processing for each individual query and response, it has non-trivial compute overheads. Thus, the centralized proxy can become bandwidth or compute bottlenecked, limiting system throughput.

[*] Equal contributions.

We present SHORTSTACK, a distributed, fault-tolerant, system for oblivious data access. SHORTSTACK achieves three desirable goals: (1) formal oblivious data access guarantee against passive persistent adversaries, even under failures; (2) system availability even when an arbitrary, bounded-sized, subset of distributed proxy servers may fail; and (3) near-linear throughput scalability with number of distributed proxy servers. In designing SHORTSTACK, we make three core contributions.

Our first contribution is to fundamentally establish security goals for oblivious data access in failure-prone deployments. Indeed, existing security models [6, 10–17, 26] do not capture failures. We introduce a formal security model and a security definition to study distributed, fault-tolerant, systems for oblivious data access under passive persistent adversaries. The model requires the classical oblivious data access guarantee [6, 10]—access patterns observed by the adversary must be independent of the input; in addition, to capture failures, the model requires this guarantee to hold under a powerful adversary that can fail an arbitrary (bounded-sized) subset of distributed proxy servers at arbitrary times. Informally, under our security definition, a scheme is considered secure if the access distribution over encrypted data objects is independent of the input distribution, even with adversarial choice and time of proxy server failures.

Our second contribution is design of a distributed, fault-tolerant, proxy architecture—SHORTSTACK—that enables oblivious data access against passive persistent adversaries, system availability (under a bounded number of failures), and near-linear throughput scalability with number of proxy servers. Simultaneously guaranteeing these three properties, especially when proxy servers can fail, turns out to be challenging: to avoid bandwidth and compute bottlenecks, multiple proxy servers must simultaneously process and send queries to the storage server; this makes it non-trivial, if not impossible, to ensure uniform random access over encrypted objects at all times (*e.g.*, right after one of the proxy server fails) without giving up on availability. The key insight in SHORTSTACK design is that obliviousness only necessitates that access patterns observed by the adversary are independent of the input; the requirement of uniform random access over all encrypted objects as in prior designs is one, but not the only, way to achieve such independence. SHORTSTACK design achieves such independence as follows. The security of oblivious data access techniques stems from "flattening" the access distribution over unencrypted (plaintext) objects to a uniform random one over encrypted (ciphertext) objects (Figure 1 (a)). As illustrated visually in Figure 1 (b), any uniform random distribution over ciphertext objects can be decomposed into multiple sub-distributions in a manner that (1) each sub-distribution is uniform random over its support; and (2) the set of objects in any sub-distribution is equal in size, disjoint, and random. Thus, if each proxy server that forwards queries to the storage server is responsible for one of the sub-distributions, even with failure of a subset of these proxy servers, the adversary observes nothing but a uniform distribution (using (1)) over a random subset (using (2)) of objects. Achieving independence, and not necessarily a uniform random access pattern, at all times is at the core of the SHORTSTACK design. In §4, we present a novel layered SHORTSTACK architecture that, using k physical proxy servers, maintains system availability with up to $(k-1)$ proxy server failures and achieves throughput a factor $\sim k$ higher than a single proxy, all while enabling oblivious data access.



(a) Oblivious data access approaches (b) Distributing oblivious data access approaches

Figure 1: The flattened distribution over all ciphertext keys in oblivious data access schemes can be expressed as a sum of distributions over disjoint subsets of ciphertext keys.

Our third contribution is a formal proof that SHORTSTACK enables oblivious data access under the above security model, and empirical evidence that SHORTSTACK can achieve near-perfect scalability with number of proxy servers (assuming storage server is not the bottleneck). We also show that SHORTSTACK gracefully handles failures: in the worst-case, SHORTSTACK throughput reduces linearly with number of proxy server failures (as one would expect). For the current SHORTSTACK implementation, the cost of achieving oblivious data access, availability and scalability is a ∼7ms increase in latency, a tiny fraction of the usual wide-area network latency. An end-to-end implementation of SHORTSTACK is available at https://github.com/pancake-security/shortstack.

2 SHORTSTACK Background

We describe our system, failure, and threat models, followed by a brief primer on oblivious data access approaches.

2.1 System, Threat and Failure Models

System model. We consider settings where applications offload data to the cloud to benefit from the many properties enabled by cloud services, *e.g.*, strong data durability and persistence, geo-replication, lower cost than provisioning dedicated and replicated storage servers, transparent handling of devices wearing out, and others. Examples of such applications include cloud-based healthcare services [9, 27–29] as well as classical applications from access pattern attack literature [6, 11]. The cloud-based storage service implements a key-value (KV) store that stores a collection of KV pairs, and support the following single-key operations: get, put, and delete. SHORTSTACK design can be applied to any data store that supports single-key read/write/delete operations.

SHORTSTACK employs the standard encryption proxy model, commonly used in encrypted data stores [6, 15, 16, 30–35]: a trusted proxy orchestrates query execution from one or more client applications; the only difference compared to previous designs is that, in SHORTSTACK architecture, the proxy is logically-centralized but physically-distributed—that is, client queries may now be routed though multiple physical proxy servers within the same trusted domain.

All network channels are encrypted using TLS. Each key k in the KV store is encrypted using a pseudorandom function (PRF), denoted by $F(k)$; each value v is symmetrically encrypted, denoted by $E(v)$. The logically-centralized proxy stores secret cryptographic keys needed for F and E, and performs encryption. Since F is deterministic, the proxy can execute all queries related to key k by sending $F(k)$ to the cloud service. Similar to many existing commercial deployments [31–35], keys and values are padded to a fixed size to avoid any length-based leakage.

Threat model. SHORTSTACK builds upon the widely-used trusted proxy threat model [11–13, 15], where one or more mutually-trusting clients execute operations on an untrusted cloud storage service via a trusted proxy; as mentioned earlier, the only difference in SHORTSTACK is that the proxy is logically-centralized but comprises physically-distributed servers. As in many prior works [6, 11, 30], we consider scenarios where the clients and the proxy servers all belong to a trusted domain. The storage service is controlled by an honest-but-curious (or, a passive persistent) adversary that observes all encrypted accesses but does not actively perform its own accesses. Since network channels are encrypted using TLS, the adversary cannot observe communications within the trusted domain, that is, the adversary cannot observe traffic between the clients and proxy servers.

We model queries to the KV store using the Pancake model [6]: queries are generated as a sequence of accesses sampled from a (time-varying) distribution π over n KV pairs. While the encryption mechanism has an estimate of the distribution $\hat{\pi}$, the adversary knows both the distribution π and the transcript of encrypted queries and responses. We define a formal security model and definition in §5, but informally, the system is secure if the transcript is *independent* of the underlying distribution π, i.e., the adversary cannot identify an association between the two.

Failure model. We assume the cloud service provides data durability. However, proxy servers can fail. We consider the fail-stop model [36] for proxy server failures.

2.2 Oblivious Data Access Approaches

There are two approaches to oblivious data access today—the classical ORAM [10–17], and the more recent approach of frequency smoothing as in Pancake [6, 18, 19]. ORAMs are designed to prevent a broad range of attacks (*e.g.*, active adversaries); accordingly, they also suffer from high overheads, *e.g.*, recent results [20–25] have established strong lower bounds on ORAM overheads—for a data store with n KV pairs, any ORAM design must incur bandwidth overheads of $\Omega(\log n)$ (for proxy storage sublinear in KV store size). For KV stores that store millions or billions of KV pairs, these overheads may amount to orders-of-magnitude of throughput loss [6,37], making ORAMs impractical. Pancake enables oblivious data access against passive persistent adversaries, and incurs a small, constant, bandwidth overhead of 3×, independent of the number of objects in the KV store. Thus, we focus on building a distributed, fault-tolerant, proxy architecture within the Pancake context. To keep the paper self-contained, we summarize the Pancake mechanisms necessary to understand the SHORTSTACK architecture.

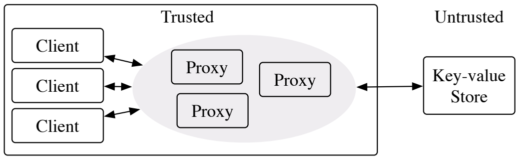


Figure 2: SHORTSTACK System and Threat model

A brief primer to oblivious data access using Pancake. The Pancake approach combines the knowledge of the distribution estimate $\hat{\pi}$ with several techniques (selective replication, fake accesses, batching, etc.) to transform a sequence of queries into uniform accesses over encrypted KV pairs. Selective replication creates "replicas" of KV pairs that have high access probability relative to other KV pairs, which serves to partially smooth the distribution over (replicated) KV pairs, while also ensuring that the total number of keys to be stored in the KV store is exactly $2n$ (if needed, dummy replicas are added so that the number of replicas does not reveal any distribution-sensitive information). To hide the association between the original keys and their replicas, each replica (k,i) of an unencrypted key k is protected by applying the pseudorandom function F, discussed in §2.1, to the replica identifier to generate an encrypted label $F(k,i)$ for the replica. In the rest of the paper, we refer to the original unencrypted key as the plaintext key, and the encrypted label for each replica as the ciphertext key. To remove the remaining non-uniformity, "fake" queries are added: these queries are sampled from a carefully crafted fake access distribution π_f to boost the likelihood of accessing replicated KV pairs, until the resulting distribution is uniform.

Security requires ensuring that fake and real queries be indistinguishable; to achieve this, encrypted queries are issued in small batches of size B, where each query is either real or fake with equal probability. Since the adversary cannot observe traffic between the clients and the proxy server, it has no way to distinguish real and fake queries within any batch. To prevent an adversary from distinguishing between reads and writes, every access is performed as a read followed by write of a freshly encrypted value. Writes to keys with multiple replicas could reveal which replicas belong to the same key; thus, only one replica is updated at the time of the write query, and the write value is cached at the proxy in a data structure called the UpdateCache, and the remaining replicas are opportunistically updated during subsequent fake or real queries to the replicas.

Dynamic adaptation to changes in the underlying access distribution is achieved by adjusting the fake-distribution (π_f), and by reassigning the number of replicas across keys. This can be done securely by exploiting the observation that the total number of replicas is exactly $2n$, regardless of the underlying distribution. As such, when the distribution changes, for every key that must lose a replica, another must gain a replica to ensure the distribution remains smooth. Thus, replicas can be reassigned opportunistically for all such key-pairs using a replica-swapping protocol.

In summary, to enable oblivious data access for the general case of read/write workloads and for time-varying distributions, Pancake uses a centralized, *stateful*, proxy that stores (1) the UpdateCache to buffer writes until they are opportunistically propagated to all the replicas; (2) distribution-related state; and (3) replica-related state, to execute the replica swapping process during distribution changes. Using this state, Pancake enables oblivious data access by performing three tasks at the proxy in failure-free scenarios: (1) generating "fake" queries for each real client query; (2) updating UpdateCache upon each query; and (3) issuing a batch of queries comprising real and fake queries to the server, and relaying the response for the real query back to the client.

3 Limitations of Strawman approaches

In this section, we describe subtle security vulnerabilities with strawman approaches to designing distributed, fault-tolerant, systems for oblivious data access.

3.1 Centralized proxy: Insecure and/or long periods of unavailability

The stateful nature of the centralized proxy makes it challenging to simultaneously achieve oblivious data access security guarantees, availability and scalability upon a failure. If achieving scalability were the only goal, the proxy server could be overprovisioned with large bandwidth and/or compute resources; however, achieving security and availability upon a failure is hard due to the proxy being stateful: the naïve solution of replacing the failed proxy server with a new one and having clients reissue failed queries results in violating security and correctness guarantees:

- Consider the (simplest) case of a read-only workload with a static access distribution. Replacing a failed proxy server with a new one, and having clients reissue the failed queries, results in the following subtle security issue. Consider a real query on key k; and consider the scenario where the proxy fails in the middle of sending out queries (both real and fake) in the batch to the KV store, that is, some of the queries in the batch have been sent out while others are lost. Since the proxy has failed, the client would receive no response for k; thus, upon restarting the proxy, the client will retry a real query on k. The retried queries will result in the same real accesses, but potentially new fake accesses. An adversary can thus exploit the transcript of queries at the server to identify real queries with high confidence by isolating repeated accesses right before and right after a failure, hence gaining sensitive information.
- Write queries make the problem significantly more challenging. Consider a write query to a key with two replicas; suppose the proxy fails when the write value has propagated to only one of the replicas (and thus, is buffered in the UpdateCache waiting to be propagated to the other replica). We now replace the failed proxy with a new one. Since the UpdateCache state is lost, when a read query for this key is received, the new proxy could end up reading the value from one of the stale replicas, violating the data correctness/consistency guarantee. Alternatively, if the new proxy reads all replicas of the key to determine which one has the latest value (*e.g.*, using timestamps), oblivious data access guarantees would be violated since an adversary can identify replica correlations (replicas being accessed belong to the same key) by analyzing queries right after a failure.

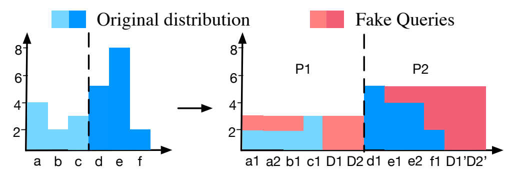
For a centralized stateful proxy design and for the general case of read/write workloads over time-varying distributions, to avoid the above security and correctness violations upon a failure, the proxy state must be reconstructed—*e.g.*, by downloading all the data from the cloud service, decrypting the data, reinitializing the data structures, re-encrypting the new data structures, and writing all the new data back to the server—before issuing new queries. Even for moderate-sized KV stores, this would incur extremely large bandwidth and compute overheads, as well as long unavailability periods.

In summary, replacing the centralized proxy server with a new one (upon a failure) and having clients reissue the queries either fails to ensure critical system properties (security and/or correctness), or results in large unavailability periods. This motivates the need for a distributed proxy architecture.

3.2 Challenges in Distributing Proxy Logic

We now describe security and correctness vulnerabilities with naïvely distributing the proxy state and logic across multiple physical servers.

Naïvely partitioning both the proxy state and the query execution responsibility leads to security violations. A straightforward approach to designing a distributed proxy for oblivious data access is to partition both the proxy state and the query execution responsibility across multiple physical servers—each proxy server stores the UpdateCache and access distribution for a subset of the plaintext keys (*e.g.*, using hash partitioning over the plaintext keys); clients forward their (real) query on key k to the proxy server responsible for k; and, upon receiving a real query, the proxy server generates fake queries based on distribution *corresponding to its own partition*, and executes these queries on the storage service.



(a) Distribution over plaintext keys

(b) Distribution over ciphertext keys

Figure 3: Security violation in one-layer approach.

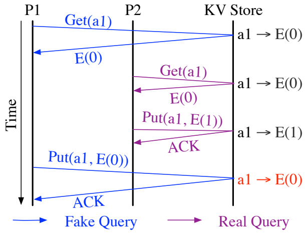
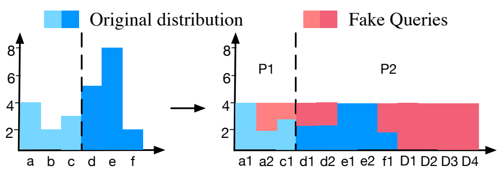


Figure 4: Correctness violation in one-layer approach.



(a) Distribution over plaintext keys

(b) Distribution over ciphertext keys

Figure 5: Security violation in two-layer approach.

While this approach scales linearly with number of physical proxy servers, it suffers from security vulnerability. In particular, it does not guarantee that the resulting distribution observed by the adversary is independent of the input distribution. Consider the scenario shown in Figure 3 (a). Here, plaintext keys are partitioned across two proxy servers—P1 is responsible for keys $\{a, b, c\}$, and P2 is responsible for keys $\{d, e, f\}$. Since, each proxy server operates only on its local plaintext key partition, P1 selectively replicates key a into 2 replicas a_1, a_2, and introduces two dummy key replicas D_1, D_2, leading to a total of 6 ciphertext keys; it then adds fake queries to make the access distribution across these ciphertext keys uniform. Similarly, P2 selectively replicates key e into 2 replicas e_1, e_2, and introduces two dummy key replicas D'_1, D'_2, again leading to a total of 6 ciphertext keys; P2 then adds fake queries to make the access distribution across these ciphertext keys uniform. Figure 3 (b) shows the final output access distribution over ciphertext keys. Since P1 and P2 smooth the distribution over their sets of plaintext keys independently, and since the key set assigned to P2 has a higher average access frequency than the key set assigned to P1, the frequency of accesses over ciphertext keys for P2 is higher than the frequency of accesses over ciphertext keys for P1. In particular, the overall access distribution over all ciphertext keys is dependent on the input distribution over the two subset of keys, thus leaking sensitive information.

Replicating proxy state across all physical servers but naïvely partitioning query execution responsibility leads to security violations. To avoid the security vulnerability in the previous scenario, one possible approach is to replicate the entire proxy state (UpdateCache and access distribution) across all physical servers in the distributed proxy. We will need to keep the state consistent across all physical servers—various mechanisms exist for this; for instance, clients can broadcast each query to each physical server to keep the access distribution consistent, and servers could use a distributed protocol (*e.g.*, state machine replication) to keep the UpdateCache consistent. Let us ignore the scalability issues with maintaining such consistent state for a moment.

To avoid bandwidth and compute bottleneck, we still want each query to be executed at one (or a small number) of the physical proxy servers. Thus, each physical server will now be responsible for receiving real queries from the clients for a subset of the keys (again, *e.g.*, using hash partitioning over the plaintext keys), and generating fake queries for each real query (now on the entire distribution). One question remains: which physical server should send the (real and fake) queries to the storage service on the cloud? Unfortunately, both the obvious solutions—the server generating the batch executes all queries in the batch, and the server responsible for plaintext key k executes all (real and fake) queries for the key k (independent of which server generated the fake query)—suffer from security and/or correctness vulnerabilities.

To see the issue with the first solution, consider the example in Figure 4 with two proxy servers P1 and P2: to serve a client query to write value 1 to key a, P2 sends a get($F(a,1)$) followed by put($F(a,1), E(1)$) query to the KV store, where $(a,1)$ is one of the ciphertext key, or replica, corresponding to a. At the same time, P1, unaware of P2 ongoing write query, sends a fake put query to the same ciphertext key $(a,1)$ in response to another client query. Based on the timeline of operations shown in Figure 4, the fake put from P1 overwrites the real put from P2, resulting in incorrect system behavior. Note that the incorrectness occurs since two different proxy servers issue queries for the same ciphertext key.

Unfortunately, the second solution also suffers from security vulnerabilities—partitioning the query execution across physical servers reveals not only which *plaintext* keys are managed by each server, but also their relative access frequencies. Figure 5 shows an example; the scenario is the same as Figure 3, but with selective replication and fake query generation done over the entire distribution across all plaintext keys—thus, as shown in Figure 5 (b), in addition to selective replication of keys d and e, 4 dummy key (D) replicas (D_1, D_2, D_3, D_4) were added, and the access distribution across ciphertext keys is uniform. We use the same partitioning of plaintext keys across P1 and P2 as in the example of Figure 3—P1 handles all real and fake queries for the three less popular plaintext keys, while P2 handles all queries for the three more popular plaintext keys and the dummy key. The challenge, however, is that although each server handles roughly equal number of plaintext keys, the number of ciphertext keys handled by P1 ($= 3$) and P2 ($= 9$) are very different. This leaks the subset of keys handled by each server and, by extension, their relative popularities to the adversary. Even if the volume of traffic issued by individual proxy servers is hidden (*e.g.*, via a trusted gateway/NAT so that all proxy servers have the same publicly visible IP address), failures of one of the physical proxy servers would reveal the same information. Even if clients were to retry their queries upon a failure, in-flight queries to the KV store from a failed server would be repeated, again revealing the same information.

Summary. The above discussion leads to three different design principles for distributed, fault-tolerant, oblivious data access systems. From the partitioning-based approach, we learn that, to achieve oblivious data access, each physical server in the distributed proxy should perform selective replication and (fake) query generation over the entire distribution across all plaintext keys (thus, each physical server should know the access distribution across the entire set of plaintext keys). The replication-based approach leads to two additional principles. First, even if proxy state can be replicated in a consistent and scalable manner, maintaining correctness requires that no two physical proxy servers should send the queries for the same ciphertext key; in other words, query execution should be partitioned by ciphertext keys across different physical servers. Second, to avoid security vulnerability, no single proxy server should be deterministically responsible for executing queries for all ciphertext keys corresponding to the same plaintext key; that is, query execution should be partitioned by ciphertext keys—randomly, and independent of plaintext keys—across physical proxy servers.

4 ShortStack Design

We now present the ShortStack distributed, fault-tolerant, proxy architecture.

4.1 Design Overview

ShortStack uses a novel layered architecture, with three *logical* layers*, as shown in Figure 6. Each layer has multiple logical proxy servers for fault tolerance and/or scalability purposes, and embodies one of the three design principles outlined at the end of the previous subsection. In the first layer (L1), proxy servers are responsible for a random subset of client queries—upon receiving a real client query on a plaintext key, the server generates real and fake queries (over ciphertext keys); importantly, fake queries are generated using the *entire* access distribution across all plaintext keys. In the second layer, L2, proxy servers are responsible for maintaining a partition of the UpdateCache state; importantly, the UpdateCache is partitioned by *plaintext* keys across the L2 servers. Finally, in the third layer, L3, each proxy server is responsible to execute real and fake queries on the KV store for a random, distinct, subset of *ciphertext* keys.

We outline the lifetime of a query with the layered ShortStack architecture in a *failure-free scenario*. The client sends the query to a randomly selected L1 proxy server; the L1 server generates the batch comprising real and fake queries (recall, these generated queries are on ciphertext keys). The L1 server then forwards each individual query within the batch to one of the L2 servers—the one that maintains UpdateCache state for the corresponding plaintext key in the query. Upon receiving a query, an L2 server updates its local partition of the UpdateCache, and forwards the query to one of the L3 servers—the one that is responsible for executing queries for that ciphertext key. The L3 server ultimately forwards the query to the KV store; upon receiving a response from the KV store, the L3 server sends a response for the real query back to the client, as well as an acknowledgement in the reverse direction of the original path taken by the query (from L3 to L2 to L1) to clear any buffered state associated with the query. We provide more details on the three-layer ShortStack architecture and query execution in §4.2.

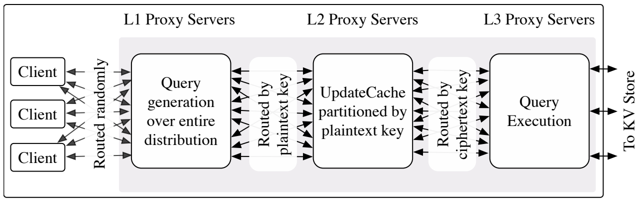


Figure 6: An overview of three-layer ShortStack architecture

For fault-tolerance against f failures, each of the L1 and L2 proxy servers use $f+1$ replication along with the chain replication protocol [39]. Replicating L1 servers prevents the security vulnerabilities discussed in §3.1 that are caused by clients retrying queries upon failures. Specifically, as we discuss in §4.3, ShortStack uses chain replication to guarantee that a batch of queries is never partially executed—either all the queries in a batch are (eventually) forwarded to the KV store or none of them are—thus preventing access pattern leakage even when failures happen. Replicating L2 servers prevents UpdateCache state from being lost due to failures. As we will show, L3 server failures do not have the same security vulnerability as L1 and L2 server failures. Thus, L3 layer is not chain replicated; however, it needs at least $f+1$ servers to maintain availability during failures—if one of the L3 server fails, the remaining L3 servers take over its load. Upon an L3 server failure, in-flight queries at the server will be dropped and L2 servers will reissue the dropped queries. While this results in duplicate queries being forwarded to the KV store, we will show that these duplicate queries do not reveal any sensitive information—the adversary would only observe duplication of queries to a random subset of labels independent of the input distribution. We provide more details on ShortStack mechanisms for handling failures in §4.3.

ShortStack design allows independently setting desired fault tolerance f and scalability factor k. Specifically, to achieve a factor k scalability—that is, achieving system throughput a factor of k higher than a centralized proxy—along with fault tolerance against f failures, ShortStack creates k independent L1 and L2 chains that operate in parallel. The case of L3 is again different; if $f+1>k$, ShortStack will already have at least k L3 proxy servers to ensure fault tolerance (as described earlier). For $f+1 \leq k$, ShortStack

[*] On a lighter note, our work seems to formally establish the widely-agreed belief that three is the right number for a ShortStack [38]!

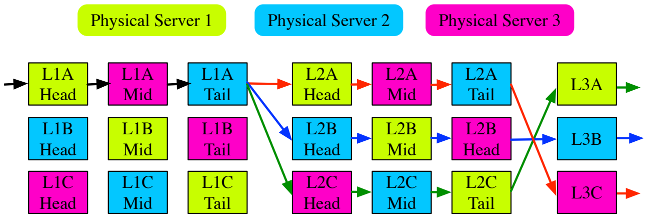


Figure 7: An instantiation of SHORTSTACK's three-layer architecture that guarantees system security and availability with $f = 2$ failures, and achieves $k = 3\times$ scalability (as defined in §4.1). Multiple logical layers in SHORTSTACK are colocated on the same physical server. The arrows depict the lifetime of a single query.

uses a total of k L3 proxy servers, thus guaranteeing both fault tolerance against f failures and a factor k scalability. Figure 7 shows an example for $f = 2$ and $k = 3$.

Colocating SHORTSTACK logical layers and their replicas on a small number of physical servers. Consider an instantiation of SHORTSTACK with fault-tolerance against up to $f = 2$ failures and $k = 3$ scalability. Then, given the above design, SHORTSTACK will require 3 L1 and 3 L2 chains, each having 3 logical replicas within the chain replication protocol. In addition, SHORTSTACK will require 3 logical servers in the L3 layer. Overall, for $f = 2$ and $k = 3$, SHORTSTACK requires 21 "logical" units. However, as shown in Figure 7, all these 21 logical units can be packed on 3 physical servers without compromising security, fault tolerance, availability and scalability. In particular, the replicas of each logical server in each layer are staggered across the physical servers such that no two replicas of the same logical server within the same layer are co-located on the same physical server. Hence, even upon failure of any two physical servers, one replica from each of the L1 servers, one replica from each of the L2 servers, and one L3 server will still be alive, ensuring security, availability and $f = 2$ fault tolerance. In general, using a technique from [40], SHORTSTACK achieves a factor k scalability and fault tolerance against $f \leq k-1$ failures, using only k physical servers. Since any system that tolerates f failures and achieves a factor k scalability must require at least $\max(f+1,k)$ physical servers, SHORTSTACK uses minimum resources to provide these properties.

We provide more details on design of each layer in the SHORTSTACK layered architecture, the mechanisms for fault tolerance, and the mechanisms for handling dynamic distributions in §4.2, §4.3 and §4.4, respectively.

4.2 SHORTSTACK Design Details

In this subsection, we describe SHORTSTACK's three-layer architecture in detail, for the case of no failures and static access distribution. We will extend this design to handle failures and dynamic distributions in §4.3 and §4.4, respectively.

In a failure-free scenario, the key challenge that SHORTSTACK addresses relative to a single proxy architecture is scalability. To achieve k-factor scalability in the failure-free scenario, SHORTSTACK uses k (logical) proxy servers in each layer. For example, in Figure 7, each layer would consist of three nodes, *e.g.*, L1A, L1B and L1C for the L1 layer, L2A, L2B and L2C for the L2 layer, and L3A, L3B and L3C for the L3 layer.

Details of three-layer operation. Figure 8 details the precise initialization and L1/L2/L3 server logic in SHORTSTACK. SHORTSTACK employs the following functionalities from PANCAKE ($\mathcal{P}$) [6] as a black-box:

- an Init function, which takes as input an estimate of the access distribution $\hat{\pi}$ and the unencrypted KV store KV of size n plaintext keys, and generates an encrypted KV store KV′ of size $2n$ ciphertext keys, along with a fake distribution π_f over KV′;
- a Batch function, which takes a query on a plaintext key k in KV as input, and generates (using $\hat{\pi}$ and π_f) a batch of B ($B = 3$ by default) ciphertext queries to KV′; and,
- an UpdateCache function that internally updates per-plaintext key state, and returns an encrypted (possibly updated) value to be written to the KV store.

We now outline how SHORTSTACK distributes the execution of PANCAKE across its three-layer design:

*Initialization (*Init() *in Figure 8):* SHORTSTACK first performs PANCAKE initialization (using $\mathcal{P}$.Init), transforming the unencrypted KV store KV with n plaintext keys to the encrypted KV store KV′ using $2n$ ciphertext keys, using an estimate of the underlying access distribution $\hat{\pi}$. During the process, the adversary just observes insert operations of $2n$ ciphertext keys, which does not reveal any information. SHORTSTACK then initializes and configures k logical proxy servers in each of the three layers on top of k physical servers. Finally, SHORTSTACK computes a weight vector δ, containing weights assigned to each L2 server (proportional to the volume of ciphertext traffic generated by it). As will be discussed, L3 servers use these weights to process L2 queries such that the queries issued by L3 servers appear uniform random (recall, this subsection focuses on failure-free scenario, where SHORTSTACK achieves uniform random distribution over ciphertext keys).

*Query processing logic (s_{L1}.*ProcessQuery()*, s_{L2}.*Process() *and s_{L3}.*Process() *in Figure 8):* Clients forward each query to a randomly chosen L1 proxy server. Upon receiving a query, the L1 server generates a batch of B queries (using $\mathcal{P}$.Batch) that comprises both real and fake queries to KV′. The L1 server then enqueues each query in the batch across different L2 servers based on the hash of the query's plaintext key.

Upon receiving a query, an L2 server calls $\mathcal{P}$.UpdateCache which leads to two sequential actions. First, the per-plaintext key state stored at the L2 server is updated; and second, if this query can be used to propagate outstanding write queries into the plaintext key replicas, the value to be written to the KV store is updated. It then forwards the query to the corresponding L3 server based on the hash of the query's ciphertext key (denoted as "Enqueue" in Figure 8).

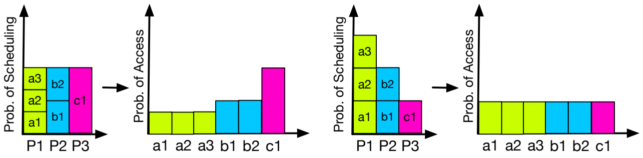
```
$\underline{\text{Init}(\hat{\pi}, \mathsf{KV}, S, f)}$:
    $\mathsf{KV}', \pi_f \leftarrow \mathcal{P}.\mathsf{Init}(\mathsf{KV}, \hat{\pi})$
    $S_{\mathrm{L1}}, S_{\mathrm{L2}}, S_{\mathrm{L3}} \leftarrow \mathsf{Configure}(S)$
    $\delta \leftarrow \mathsf{Weights}(S_{\mathrm{L2}}, \mathsf{KV}')$
    return $\mathsf{KV}', (S_{\mathrm{L1}}, S_{\mathrm{L2}}, S_{\mathrm{L3}}), \delta$

$\underline{s_{\mathrm{L1}}.\text{ProcessQuery}(k, v)}$:
    $\ell \leftarrow \mathcal{P}.\text{Batch}(k)$
    For $((k, j), v) \in \ell$:
        $s_{\mathrm{L2}} \leftarrow S_{\mathrm{L2}}[\mathcal{H}(k)]$
        $s_{\mathrm{L2}}.\text{Enqueue}((k, j, v))$

$\underline{s_{\mathrm{L2}}.\text{Process}()}$:
    $k, j, v \leftarrow \text{Dequeue}()$
    $v \leftarrow \mathcal{P}.\mathsf{UpdateCache}(k, j, v)$
    $s_{\mathrm{L3}} \leftarrow S_{\mathrm{L3}}[\mathcal{H}(F(k, j))]$
    $s_{\mathrm{L3}}.\text{Enqueue}(s_{\mathrm{L2}}, (F(k, j), v))$

$\underline{s_{\mathrm{L3}}.\text{Process}(\delta)}$:
    $s_{\mathrm{L2}} \overset{\delta}{\leftarrow} S_{\mathrm{L2}}$
    $k', v \leftarrow \text{Dequeue}(s_{\mathrm{L2}})$
    $v \leftarrow \text{ReadThenWrite}(\mathsf{KV}', k', v)$
    return $k', v$
```

Figure 8: **SHORTSTACK initialization and processing logic at L1, L2 and L3 servers.** $S_{\mathrm{L1}}, S_{\mathrm{L2}}, S_{\mathrm{L3}}$ are the sets of proxy servers in each layer, and S is the set of physical servers upon which they are initialized. (k, v) corresponds to the plaintext key-value pair, while j is the replica identifier for a given replica of the key. F is a secretly keyed pseudorandom function and $\mathcal{H}$ is a consistent hash function.



(a) Round robin scheduling of queires at L3 queues results in non-uniform distribution

(b) Weighted scheduling at L3 queues ensures uniform distribution

Figure 9: Query-scheduling at L3 layer should ensure uniform distribution over ciphertext keys for security. In each figure, (left) shows the probability of scheduling queries from each of the L2 servers, while (right) shows the resulting distribution across ciphertext keys.

Finally, each L3 server maintains a separate queue for each L2 server it receives queries from, and dequeues queries from the queues following a biased distribution determined by the weight vector δ. To hide whether the query is a read or a write, SHORTSTACK employs the standard approach from prior oblivious data access schemes of performing each queries as a read followed by a write to the KV store. Specifically, in Figure 8's ReadThenWrite() method, the L3 server first reads and decrypts the value associated with the query from the key-value store. If the value needs to be updated (i.e., write query), then the plaintext value is updated accordingly. Finally, the L3 server writes the (re)encrypted value for the query back to the KV store.

Query scheduling at L3 layer for security. The way in which queries from different L2 servers are scheduled at each L3 server has security implications. As a concrete example, consider a scenario where three plaintext keys a, b and c with 6, 4 and 2 replicas (or, ciphertext keys), respectively, are mapped to three different L2 servers P_1, P_2 and P_3. Suppose we have two L3 servers, and one of these handles half of the ciphertext keys for each plaintext key (Figure 9 illustrates the example, focusing only on one of the L3 servers and the ciphertext keys mapped to this server). If the L3 server processes queries from each server with equal likelihood (*e.g.*, using round-robin scheduling), then the distribution across ciphertext keys would no longer be uniform, since queries from the first server would be under-sampled while those from the third server would be over-sampled (Figure 9 (a)). To ensure L3 servers still issue queries that are uniform random, they maintain a separate query queue for each L2 server, and process the queues in proportion to the volume of traffic the corresponding L2 servers generate. In the above example, the L3 server would schedule queries from each of the L2 servers with probabilities $3/6$, $2/6$ and $1/6$, respectively, leading to a uniform distribution across ciphertext keys (Figure 9 (b)).

Accurately estimating the access distribution. SHORTSTACK employs a lightweight mechanism through which a single L1 proxy server can observe all client queries, enabling distribution estimation as accurately as a centralized proxy system [6]. One of the L1 proxy servers, designated as the leader, monitors the access distribution (handling failures will be discussed in the next subsection). Upon receiving a query, an L1 proxy server asynchronously forwards the corresponding plaintext key—and not the entire query—to the L1 leader, ensuring that the leader has a complete view of the access distribution. Sending the plaintext key and not the entire query to the leader is an useful optimization for both read and write queries—it reduces the additional network load (for write queries, values are not forwarded; for read queries, the responses are not forwarded) since the plaintext key is typically much smaller than the value itself (*e.g.*, 8B keys for 1KB value in [41]). As such, this has negligible impact on SHORTSTACK scalability and performance.

4.3 Handling Failures

We now describe how SHORTSTACK ensures fault-tolerance while preserving security and correctness under failures. We assume the standard *fail-stop* failure model [36, 39]. SHORTSTACK employs a separate centralized *coordinator* node which keeps track of the health of the proxy servers using heartbeats, detects failures, and notifies other proxy servers as needed to designate a fail-over node. The coordinator node is also replicated using ZooKeeper [42] for strong consistency. As such, a $(2r+1)$-replicated coordinator can tolerate up to r failures without any security or performance consequences.

Handling L1 and L2 failures. Failure of a single L1 server does not impact the availability of SHORTSTACK, as future client queries could potentially be load balanced across the remaining L1 servers. Such a failure, however, has security implications—consider the case where an L1 proxy server fails in the middle of forwarding a batch of queries, i.e., some of the queries in the batch have been forwarded, but others are lost due to the failure. Any real queries that are lost would need to be retried by clients. The retried queries would now result in the same real accesses, but with *new* fake accesses generated. This permits an adversary to identify real queries with high confidence by simply isolating the repeated accesses due to failures. To protect against such a vulnerability, SHORTSTACK ensures the following invariant:

Invariant 1 (Batch atomicity)**.** *Either all of the queries in a batch are forwarded to the KV store, or none of them are.*

SHORTSTACK achieves this by replicating the state of the L1 proxy servers across multiple replicas ($f+1$ replicas to tolerate up to f failures) using chain replication protocol [39]. As shown in Figure 7, SHORTSTACK maintains staggered chains across a fixed pool of physical servers, such that each physical server hosts the head node of a single chain. Chain replication ensures that all L1 replicas in the chain buffer a batch of queries, before the queries are forwarded to L2 servers—the buffered batches are only cleared when all corresponding acknowledgements are received from the L2 servers. As such, as long as any L1 replica in the chain is online, the set of buffered batches is available and can be used to retry queries as required, ensuring Invariant 1.

Since L2 servers store UpdateCache partitions, ensuring fault-tolerance for them is crucial for availability, correctness and security. As such, SHORTSTACK replicates the UpdateCache state for any key across multiple L2 proxy replicas using chain replication, similar to L1 servers.

Within each chain of L1 and L2 servers, failures of replicas are handled as per the standard chain replication protocol [39]. Since the L1 server chains interact with the L2 server chains, additional failure handling is necessary in certain cases. Consider the interaction between an L1 tail and an L2 head: if the L1 tail fails, its predecessor in the chain becomes the new tail, and resends the queries in the buffered (unacknowledged) batches to the corresponding L2 head replicas. The L2 servers, on the other hand, discard the queries that they have already seen and forward the remaining down the chain. SHORTSTACK facilitates the detection of duplicate queries by assigning unique sequence numbers to each query. If an L2 head fails, on the other hand, its successor become the new head. All L1 tails then examine their buffered batches to resend queries that were destined to the failed L2 head. As before, the new L2 head simply discards any queries that it has already seen, forwarding the remaining down the chain.

Handling L3 failures. Unlike L1 and L2 servers, L3 servers are not replicated, and hence entail different failure handling. Since L3 servers are stateless, if an L3 server fails, the remaining L3 servers can assume the responsibility of the ciphertext labels that the failed server was handling. Since the system remains available as long as at least one of the L3 servers is online, we need at least $f+1$ L3 servers to tolerate f failures. However, there are two subtle issues that can arise due to L3 failures—we describe these next, along with how SHORTSTACK addresses them.

On an L3 server failure, queries that were in-flight at the failed L3 server would be lost, which can then be retried by the L2 servers. Note that such retries can cause duplicate queries being sent to the KV store. Since the duplicate queries are to uniformly accessed ciphertext keys, it may seem like they do not reveal any distribution-sensitive information. However, repeating the queries in exactly the same order (or a correlated order) introduces a subtle security vulnerability. Specifically, when an L3 server fails, L2 tail servers repeat buffered queries (which are uniform random) and redistribute them to different L3 servers. If the order of these queries is exactly the same as before, an adversary can identify the sequences of repeated queries and correlate them to the L2 server that generated those queries. Moreover, the adversary can also map the specific ciphertext keys corresponding to the plaintext keys managed by a particular L2 server, revealing distribution sensitive information. To prevent this leakage, SHORTSTACK *randomly shuffles* buffered queries before repeating them—we formally prove in Appendix A how this ensures security under L3 server failures.

Recall that the L3 server performs a read followed by a write for all queries. For read queries (fake or real), the write simply writes back the value read from the KV store, i.e., a *fake* write. This can lead to consistency issues during failure of L3 servers—fake in-flight write queries sent by a failed L3 server prior to failure could be delayed by the network and overwrite a real write query sent by the new L3 proxy server responsible for the same ciphertext key. To address this issue, after an L3 failure, the L2 servers delay repeating buffered queries for a fixed amount of time to allow potential in-flight queries from the failed L3 server to get delivered to the KV store. We select the wait time at L2 servers long enough to ensure *all* in-flight queries are propagated to the KV store.

4.4 Handling Dynamic Distributions

Designing distributed, fault-tolerant, oblivious data access systems is challenging when underlying distribution can change over time. We outline two reasons. First, the centralized proxy design (§2.2) relies on having a complete view of the underlying distribution to detect and to react to distribution changes. Detecting the change when queries are spread across multiple proxy servers, and informing other proxy servers about the same, introduces the first challenge. Second, if different proxy servers independently initiate and terminate the replica swapping phase at different times, the resulting distribution may not appear uniform random to an adversary. As such, the adversary may be able to leverage this information to identify the keys that may have changed in popularity. We next discuss how SHORTSTACK resolves these challenges.

To detect distribution changes, SHORTSTACK leverages the L1 leader, which has visibility of all client queries (§4.2). The L1 leader is responsible for monitoring the access distribution and employs standard statistical tests to check if there is a change in distribution (i.e., from $\hat{\pi}$ to $\hat{\pi}'$) similar to PANCAKE. Upon detecting a change in distribution, the L1 leader initiates the distribution change process. To ensure security and correctness during distribution change, the L1 leader employs a specialized protocol inspired by two-phase commit (2PC) [43] to facilitate an *atomic* transition from $\hat{\pi}$ to $\hat{\pi}'$ across all servers in its three-layer design, both during the initiation and termination of the replica-swapping phase employed by PANCAKE. Our 2PC-based approach guarantees:

Invariant 2 (Distribution change atomicity)**.** *Once any* L3 *proxy server issues a query according to* $\hat{\pi}'$*, all subsequent queries issued by any* L3 *server must be according to* $\hat{\pi}'$.

In other words, there is an instant of time t_c in the protocol's execution, such that: (1) before t_c, all queries are processed according to the distribution $\hat{\pi}$, and (2) after t_c, all queries are processed using $\hat{\pi}'$. This allows us to ensure security for SHORTSTACK even under dynamic distributions, as we detail in §5. The invariant also ensures consistency during distribution change. In particular, since the change of distribution can result in a change in number of replicas for various plaintext keys, the invariant ensures queries from old and new distributions are not mixed together; this guarantees consistency by ensuring stale replicas from the old distribution are not updated incorrectly due to the new distribution by different L2 proxy servers. We show that our protocol guarantees the above invariant, with a precise specification in Appendix B. Failures during the above protocol are handled transparently by chain replication as L1, L2 servers are chain replicated. This ensures that even with failures during protocol execution, Invariant 3 is still preserved. As demonstrated in §6, SHORTSTACK can recover from such failures quickly enough so as to ensure that their effects are not perceptible to an adversary.

5 Security Analysis

This section presents a security model for access pattern attacks on a system with distributed, *fault-tolerant* proxy servers, and a proof that SHORTSTACK achieves security under this model.

5.1 Need for New Security Definitions

State-of-the-art ROR (real-or-random indistinguishability) based security definitions for access pattern attacks [6] are unable to capture the security implications of our distributed proxy setting due to two main reasons. First, ROR-based definitions focus on indistinguishability between a real and a uniform random distribution (over the entire support). However, as discussed in §3.2, we do not yet know whether it is possible to guarantee uniform random distribution over the entire support during failures for *any* distributed proxy architecture. Our IND-based security model and definitions capture the powerful intuition that uniform random distribution is not even necessary: even though the distribution is non-uniform under failures, the adversary does not gain any *usable* advantage as long as the final distribution is independent of the real distribution. More precisely, our IND-based security focuses on demonstrating indistinguishability between two arbitrary input distributions. As we will show, under our model, the only information revealed to the adversary is that a failure occurred, information the adversary already possess; it cannot, however, use this information in inferring any information about the underlying distribution itself. While it is not uncommon for IND security to reduce to ROR security in many settings, this is clearly not the case in our setting if (and, as we note later, only if) there are failures.

```
$\text{IND-CDFA}^{\mathbb{A}}_{b,q,S,f,\pi_0,\hat{\pi}_0,\pi_1,\hat{\pi}_1}$:
$\mathsf{KV}, \mathcal{T}, \mathsf{st}_A \leftarrow\!\!\$\; \mathbb{A}_1(f, S)$
$(\mathsf{KV}', \mathcal{C}, \delta) \leftarrow\!\!\$\; \mathsf{Init}(\hat{\pi}_b, \mathsf{KV}, S, f)$
For $i$ in 1 to $q$:
    $w \leftarrow\!\!\$\; \pi_b$
    $W \leftarrow W \cup \{w\}$
$\tau_1, \tau_2, \ldots \leftarrow \mathsf{Process}(W, \mathcal{C}, \mathcal{T}, \mathsf{KV}', \delta)$
$b' \leftarrow\!\!\$\; \mathbb{A}_3(\mathsf{st}_A, \mathsf{KV}', \tau_1, \tau_2, \ldots)$
**return** $b'$
```

Figure 10: **IND-CDFA security game.**

The second reason for needing new security model and definitions is that ROR-based definitions fail to capture the impact of query reordering on the transcripts observed by an adversary due to (i) distributed query processing, and (ii) worst-case timings of proxy failures. Specifically, a key challenge in demonstrating security lies in precisely capturing the effect of the distributed and failure-prone execution of any scheme in a sequential game-based proof, which the ROR-based approach omits. We thus have to develop accurate *simulators* that transform distributed query processing to an equivalent sequential one. Our model and definitions are not specific to SHORTSTACK, and can be used as templates for any distributed, fault-tolerant, proxy design.

When there are no failures, our security definition captures the same security guarantees as prior work [6] — our extensions to the model are required to capture the effect of failures in the distributed proxy setting. In incorporating these extensions, we have only strengthened the adversary.

5.2 Security Definitions and Proof of Security

We call our security definition Indistinguishability under Chosen Distribution and Failure Attack, or IND-CDFA (Figure 10). The game IND-CDFA is parameterized by bit b (to pick one out of the two given distributions), number of queries q, the set of proxy servers S on which the distributed oblivious data access protocol runs, the maximum number of server failures f allowed (similar to classical distributed systems literature that provides fault tolerance up to a fixed number of failures), and two distributions (and their estimates) that the adversary tries to distinguish between.

The adversary first outputs KV pairs KV and a queue $\mathcal{T}$ of at most f failure events. Each failure event e is characterized by the tuple (n, t, γ, r), where n is the server in S that fails, t is the time at which the last query is issued by n before failure, $t - \gamma$ is the time at which the last query was acknowledged at n before failure, and r is the failure recovery time. Next, the distributed proxy scheme's Init function generates transformed KV pairs KV', a set of (potentially replicated) servers $\mathcal{C}$, and internal state δ specific to the scheme. For instance, in SHORTSTACK, $\mathcal{C}$ consists of two sets of replicated server chains (with replication factor $f+1$) corresponding to L1 and L2 layers, and a set of $> f$ unreplicated servers for the L3 layer. The state δ corresponds to weights for L3 servers used in query scheduling, as outlined in §4.2.

After initialization, q queries are drawn from the distribution π_b and populated into the vector W. The proxy scheme's Process function takes W, $\mathcal{C}$, $\mathcal{T}$, KV' and δ as input, and generates the output transcripts τ, which is fed to the adversary to try and guess the underlying distribution (i.e., the bit b). The adversary "wins" if it guesses b correctly. Intuitively, the security goal captured by the definition rules out access pattern attacks since the probability of accessing an encrypted label in KV' is independent of the underlying distribution itself, and an adversary cannot determine which distribution was used to generate accesses to KV'.

Note that, IND-CDFA definition is independent of SHORTSTACK's design. Specifically, our definitions only assume the presence of multiple failure-prone proxy servers which are initialized using an Init function and process queries using a Process function, neither of which are specific to SHORTSTACK. Thus, our security model and definitions can be used to study oblivious data access properties of any distributed system that can factor its initialization and query processing logic along these two functions.

The following theorem establishes the security of SHORTSTACK under IND-CDDFA:

Theorem 1 (IND-CDFA Security)**.** *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi_0, \hat{\pi}_0, \pi_1, \hat{\pi}_1$ be query distributions. For any q-query* IND-CDFA *adversary $\mathbb{A}$ against* SHORTSTACK *there exist adversaries $\mathbb{B}$, $\mathbb{C}$, $\mathbb{D}_1$, $\mathbb{D}_2$ such that*

$$\mathbf{Adv}^{\textit{ind-cdfa}}_{\text{SHORTSTACK}}[(\mathbb{A})] \leq \mathbf{Adv}^{prf}_{F}[(\mathbb{B})] + \mathbf{Adv}^{ror}_{E}[(\mathbb{C})] + \mathbf{Adv}^{dist}_{Q,\pi_0,\hat{\pi}_0}[(\mathbb{D}_1)] + \mathbf{Adv}^{dist}_{Q,\pi_1,\hat{\pi}_1}[(\mathbb{D}_2)]$$

where F, E are PRF, AE schemes used by SHORTSTACK*. Adversaries $\mathbb{B}, \mathbb{C}, \mathbb{D}_1, \mathbb{D}_2$ run in same time as $\mathbb{A}$ with Q queries.*

Our security proof stems from three key components:

- Security of E as a randomized authentication scheme applied over values and F as a pseudorandom function applied over keys; this is rigorously analyzed in prior work [44, 45].
- Our estimate $\hat{\pi}$ of the underlying distribution π is sufficiently accurate. While this estimate may not be perfect, our security model only requires that $\hat{\pi}$ and π be indistinguishable for a limited number of samples, which holds for estimators used in prior work [6] on real-world workloads [41]. Since our design employs a single leader L1 server to estimate the underlying distribution using the keys for all client queries (§4.2) and employs the same estimators as prior work, its estimation is just as accurate.
- Accesses issued to the KV store reveal nothing about the underlying distribution π.

To prove the third component, we introduce simulators to sequentialize the distributed execution of SHORTSTACK's query processing to make it compatible with our game-based definition (Figure 10). Specifically, we simulate Process function for SHORTSTACK by first generating the intermediate transcript, β, assuming no failures. We do so by (i) going layer by layer and executing processing logic at appropriate servers in SHORTSTACK, and (ii) incorporating the impact of network reordering across queries between layers. We then use a Transform simulator to capture the effect of failures and generate the final transcripts τ from β. We do so by recursively applying the effect of L3 server failure events in $\mathcal{T}$ on the intermediate transcripts β in the order that they occur.

Finally, we show that the final transcripts τ are independent of intermediate transcripts β, and then show that β are independent of the underlying distribution π. The first part holds since SHORTSTACK randomly shuffles buffered queries before replaying them post failure (§4.3) and failure recovery time in SHORTSTACK is short enough to not be visible to an external observer given our failure model (§4.3) and as shown empirically in (§6.2). The second part holds, since the underlying oblivious data access scheme [6] in SHORTSTACK generates uniform random queries (§4.2) and network reorderings between layers are independent of π.

Finally, to model dynamic distributions, we generalize the above definition to Indistinguishability under Chosen Dynamic Distribution and Failure Attack or IND-CDDFA. This definition, along with the proof of SHORTSTACK's security under it, formal descriptions of our simulators, and the proof for independence of τ and π are presented in Appendix A.

6 Evaluation

SHORTSTACK is implemented in $\sim$ 6k lines of C++, using Thrift as the RPC library, AES-CBC-256 for encrypting values, HMAC-SHA-256 as our PRF, and Redis as the KV store.

Compared systems. We compare SHORTSTACK performance against two baselines. The first baseline is distributed, but encryption-only, that is, it encrypts data and client queries, but does not guarantee oblivious data access; here, client queries are randomly load balanced across stateless proxy servers that perform encryption/decryption and forward queries to the KV store. This baseline serves as an upper bound on the performance that can be achieved by any oblivious data access system (including SHORTSTACK). The second baseline is a centralized PANCAKE [6] proxy server. While this suffers from security and availability problems in the face of failures (§3.1), its performance serves as a reference point for understanding SHORTSTACK's scalability.

Experimental setup. We run our experiments on Amazon EC2. By default, we host the proxy instances across c5.4xlarge VMs with 16 vCPUs (8 cores with 2 threads per core), 32 GB RAM and 10Gbps network links. In order to emulate a cloud KV store with practically infinite bandwidth,

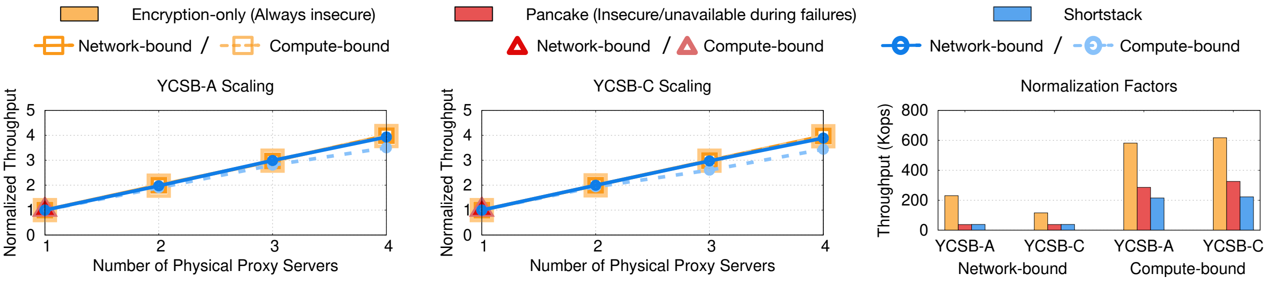


Figure 11: **Scalability properties of different systems when network bandwidth and compute are the bottleneck.** (left, middle) show system throughput normalized by throughput for a single physical proxy server, while (right) shows system throughput with a single physical proxy server. The Encryption-only lines for the network-bound and compute-bound cases overlap, since its throughput scales linearly in both cases. Since Pancake is centralized, it only has a single data point at $X = 1$ for each of the cases, and these points overlap. See §6.1 for details.

we use a single powerful VM, c5d.metal (96 vCPUs, 128 GB RAM) with large network bandwidth (25 Gbps). Similar to prior work [6], we emulate WAN access link bandwidth by throttling the bandwidth from each proxy server to the KV store server to 1Gbps. The clients run on lightweight t3.2xlarge VMs (8 vCPUs, 32GB RAM) in the same LAN. Both PANCAKE and SHORTSTACK use a batch size of $B = 3$.

Dataset and Workloads. We use the standard YCSB benchmark [41] to generate our dataset and workloads. The dataset comprises 1 million KV pairs, with 8B keys and 1KB values. We use workloads A (50% reads, 50% writes) and C (100% reads) for our experiments. YCSB workloads perform accesses distributed according to the Zipfian distribution [41]; unless otherwise stated, the skewness parameter for the Zipfian distribution in our experiments is set to the YCSB default of 0.99 (that is, heavily skewed), which is representative of many real world workloads. We also perform sensitivity analysis against distribution skew.

6.1 Scalability Analysis

We now analyze SHORTSTACK's scalability with varying number of physical proxy servers under different workloads.

Throughput scaling under bandwidth bottleneck. We study throughput scaling for SHORTSTACK by varying the number of physical proxy servers and comparing its performance against the baselines. For SHORTSTACK, k physical proxy servers constitute k chain-replicated L1 instances with $\min(k,3)$ replicas each, k chain-replicated L2 instances with $\min(k,3)$ replicas each, and k unreplicated L3 instances (i.e., the system can tolerate up to $\min(k,3) - 1$ failures). For the encryption-only baseline, a separate proxy instance is run on each physical proxy server, and the PANCAKE baseline always uses only one physical proxy server.

Figure 11 shows the scalability results for two cases: one where the physical proxy servers are network-bound (solid lines), and another where they are compute-bound (broken lines). We begin with the former case; we see that SHORTSTACK throughput scales linearly with the number of physical proxy servers. Note that we normalize each system's throughput by its throughput with a single physical proxy server — Figure 11 (right) shows normalization factors for each system, i.e., throughput with single physical proxy server. The red cross shows the throughput of the PANCAKE baseline (38 KOps): SHORTSTACK's distributed design enables linear throughput gains relative to PANCAKE via scaling. The insecure baseline also scales linearly due to random load-balancing across its proxy instances. Since all proxy servers are network bound, SHORTSTACK incurs only a constant overhead (corresponding to the relative bandwidth increase due to the oblivious data access protocol) compared to the encryption-only baseline for all configurations as we scale the number of physical proxy servers. For the YCSB-C workload, the gap between SHORTSTACK and Encryption-only baseline throughput stems from the 3× overhead imposed by the PANCAKE protocol for a batch size of $B = 3$. For the YCSB-A workload, however, the encryption-only baseline throughput is 6× higher than SHORTSTACK since it can exploit the bidirectional bandwidth to the KV store for 50% reads and 50% writes. SHORTSTACK, however, already issues a read followed by a write for every query, so it is unable to similarly exploit the bidirectional bandwidth. Since YCSB-A has equal proportion of read and write queries, this situation corresponds to the worst-case bandwidth increase (6×) for SHORTSTACK relative to the encryption-only baseline.

Throughput scaling under compute bottleneck. We now analyze throughput scaling when the physical proxy servers are compute-bound: we re-run the same experiments as above, but using c5.metal EC2 VMs (96 vCPUs, 192GB RAM, 25Gbps network bandwidth) for all systems *without* throttling the access link bandwidth to the KV store server. As the broken lines corresponding to the compute-bound case in Figure 11 show, with a single physical proxy server SHORTSTACK achieves slightly lower throughput than PANCAKE for both workloads. This is because, under a compute bottleneck, SHORTSTACK incurs additional RPC processing overheads for communication between its layers. SHORTSTACK's throughput increases significantly with more physical proxy servers, achieving $3.4 - 3.6\times$ higher throughput with 4 physical proxy servers. The increase in throughput is not perfectly linear, since workload skew results in load imbalance at the L2 layer. This effect

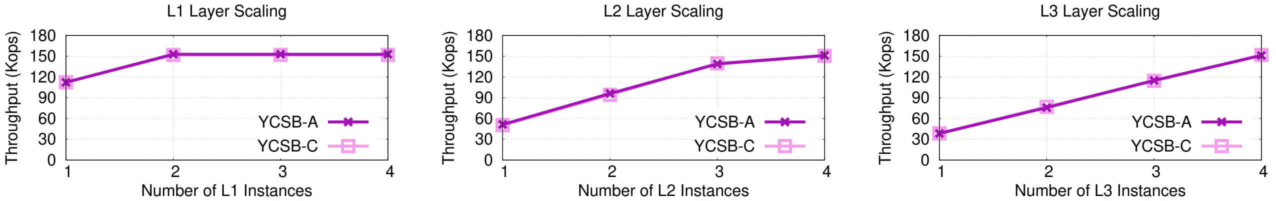
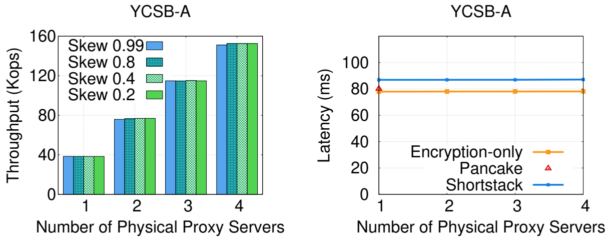


Figure 12: **SHORTSTACK layer-wise scaling for YCSB workloads A and C.** See §6.1 for details.



(a) SHORTSTACK throughput is unaffected by access skew.

(b) Query latency vs. number of physical proxy servers.

Figure 13: **SHORTSTACK throughput scaling with varying skew (a) and SHORTSTACK latency overheads (b).** See §6.1 for details.

is not observed for the network-bound case, since the network bandwidth between L3 instances and the KV store is bottlenecked before workload skew causes compute at the L2 layer to become bottlenecked. For the reminder of our evaluation, we use the network-bound setting as our default configuration.

Understanding per-layer scalability bottlenecks. Our experiments in Figure 11 scale up all layers of SHORTSTACK in equal proportions as the number of physical proxy servers are increased. To better understand SHORTSTACK's bottlenecks, we now study scalability on a per-layer basis. Since each layer performs a different component of PANCAKE logic (§4), varying the scale of each layer independently while keeping the scale of the other two layers fixed allows us to understand which step becomes a throughput bottleneck before the others. For this, we use a setup similar to Figure 11. To understand L1 layer scalability, we fix the number of physical proxy servers to 4, the number of replicated L2 instances and unreplicated L3 instances to the default (4), and vary the number of replicated L1 instances from $1-4$. We perform similar experiments for the L2 and L3 layers as well. Figure 12 shows the corresponding results for the YCSB-A and YCSB-C workloads. For the L1 layer, throughput increases slightly from $X = 1$ to 2, beyond which it saturates, since L1 is no longer the bottleneck. For the L2 layer, from $X = 1$ to 3 throughput increases, albeit non-linearly due to plaintext key-based partitioning — while the number of plaintext keys handled by each L2 server is roughly equal, the number of replicas handled by them is skewed due to the skew in the YCSB workload. At $X = 4$, the L2 layer is no longer the bottleneck. For the L3 layer, throughput scales linearly from $X = 1$ to $X = 4$ due to ciphertext key-based partitioning, with each L3 proxy handling roughly the same number of ciphertext keys.

As expected, the bottlenecks are different at different SHORTSTACK layers. When all layers are sufficiently provisioned, SHORTSTACK is able to saturate the access link bandwidth between the L3 layer and the KV store. Reducing the number of L1 and L2 proxy instances, however, leads to *compute* becoming the bottleneck at the respective layers. One of the key contributors of compute overheads are serialization/deserialization for network queries. Finally, layer-wise scaling characteristics are similar for YCSB-C and YCSB-A workloads, as UpdateCache processing in YCSB-A due to writes does not account for much of the compute overheads.

Throughput scaling with skew. We evaluate SHORTSTACK scaling for workloads with different skew for a setup similar to Figure 11. We vary the skew parameter for YCSB's Zipf distribution from 0.2 (close to uniform) to 0.99 (heavy skew) to consider both extremes. We only show our results for YCSB-A in Figure 13(a), since results for YCSB-C were similar. SHORTSTACK system throughput scales linearly regardless of skew, because the bottleneck in the end-to-end query execution is the access link bandwidth between the L3 layer and the KV store for all scales. Since the skew only affects processing at L2 layer (which is not the bottleneck), our throughput is independent of skew. While SHORTSTACK throughput scales linearly even for heavily skewed workloads, there could indeed be rare extreme-case scenarios where such would not be the case, *e.g.*, if all popular plaintext keys get consistently hashed to a single L2 instance, resulting in a compute bottleneck at that instance.

SHORTSTACK Latency overheads. To quantify SHORTSTACK's latency overheads, we evaluate end-to-end query latency for varying number of physical proxy servers for compared systems using a setup similar to Figure 11 with one change: we separate the KV store and physical proxy servers by the WAN. Figure 13(b) shows the results; again, we only show YCSB-A workload results, as YCSB-C results are similar. Independent of the scale, SHORTSTACK increases query latency by a modest 8% (additional 6.8ms) compared to PANCAKE. This increase in latency is due to additional processing and network hops introduced by SHORTSTACK's multiple layers and chain replication within the L1 and L2 layers. Nevertheless, these overheads are masked by the significantly larger WAN access latency.

6.2 Failure Recovery

We now evaluate SHORTSTACK's ability to recover from failures and also validate our assumptions in proving SHORTSTACK security. We fix the number of physical proxy servers

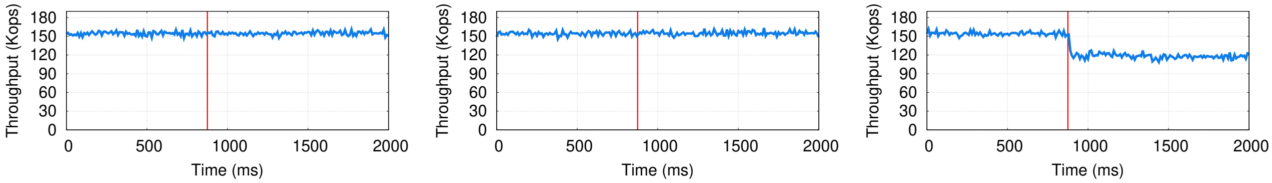


Figure 14: **ShortStack failure recovery for (left) L1, (middle) L2, and (right) L3 failures.** See §6.2 for details.

to 4, the number of 3×-replicated L1, 3×-replicated L2, and unreplicated L3 instances to 4 each, and use the YCSB-A workload. To understand the impact of failures on each layer independently, we fail one proxy instance in a particular layer by killing its associated process; for L1 and L2, we kill an arbitrary replica from one of the instances. We measure the instantaneous throughput of our system during each experiment at 10ms granularity; when measured at finer-grained timescales, we found that the instantaneous throughput numbers were too noisy to discern any meaningful trends.

Figure 14 shows the effect of failure at each layer on ShortStack throughput. We find that failures in L1 and L2 proxy chains do not cause any noticeable dip in the throughput, since ShortStack can quickly recover from failures within 3–4 ms — much faster than the average query latency over WAN ($\sim$ 90ms), and smaller than the typical variance in query latencies. Hence, an adversary cannot reliably distinguish between a failure event and variations in instantaneous throughput due to noise caused by network delays, independent of the timescale at which measurements are done. This validates our assumption for ShortStack security under failures discussed in §5 — specifically, L1 and L2 failures have an imperceptible impact on an adversary's observed access pattern to the KV store. Upon an L3 proxy failure, the throughput reduces by 25% – commensurate with the reduction in the bandwidth to the KV store server; however, since L3 layer partitions queries by ciphertext keys, it does not reveal any information about the client access patterns.

7 Related Work

We now discuss the works most closely related to ShortStack's goals of distributed, fault-tolerant, oblivious data access. ORAM [10] approaches have been adapted to real world cloud storage [12–17, 26], with recent efforts enabling *concurrency* and *asynchrony*. Oblivious Parallel RAM (OPRAM) [12, 14, 46–49] permits multiple concurrent clients to query the storage, but requires cross-client coordination per-query (*e.g.*, using oblivious aggregation [12]) to ensure no two clients concurrently issue a query for the same data. This severely limits throughput scaling under high query traffic due to compute bottlenecks.

CURIOUS [16] and TaoStore [15] employ a centralized proxy model, but permit client parallelism via *asynchrony*. Since each operation requires updates to per-plaintext key proxy state for multiple random KV pairs, extending their design to a distributed and secure one is challenging. The latest in this line of work, ConcurORAM [17] and Snoopy [26], permit multiple parallel clients to query a cloud-hosted ORAM *without* inter-client or proxy based coordination. ConcurORAM achieves this by offloading much of the synchronization to the cloud, which not only requires non-trivial changes to cloud storage, but also limits system throughput under high load. Concurrent to our work, Snoopy builds a distributed oblivious data access system (for ORAM-based designs); however, Snoopy does not prove security for scenarios where servers can fail. In any case, ShortStack and Snoopy offer the same trade-offs as discussed in [6]—Snoopy can handle active adversaries, but also incurs significantly higher overheads relative to ShortStack. Prior work [6] has empirically shown that state-of-the-art single proxy ORAM schemes achieve 220× lower throughput than Pancake for the same workloads as in our evaluation. Since ShortStack can scale Pancake's throughput linearly (§6) with number of proxy servers, even if one could design a distributed ORAM system that scales near-perfectly with number of proxy servers, the maximum achievable throughput would still be orders of magnitude lower than ShortStack.

8 Conclusion

Existing systems for oblivious data access rely on a centralized, stateful, proxy to coordinate queries between applications and the storage server. We have demonstrated that, in failure-prone deployment, such systems can suffer from security violations, long periods of unavailability and/or scalability limits. Our core contribution is ShortStack, a distributed, fault-tolerant and scalable system for oblivious data access. Using a novel layered architecture, ShortStack achieves the classical obliviousness guarantee—access patterns observed by the adversary being independent of the input—even under a powerful passive persistent adversary that can force failure of arbitrary (bounded-sized) subset of proxy servers at arbitrary times. We also introduce a security model to study oblivious data access with distributed, failure-prone, servers.

Acknowledgements

We would like to thank our shepherd, Alex C. Snoeren, and the anonymous OSDI reviewers for their insightful feedback. We would also like to thank Thomas Ristenpart for many useful discussions during this work. This research was supported in part by NSF awards 2054957, 2047220, 2118851, 1704742, Faculty Research Awards from Google and NetApp, and an IC3 fellowship thanks to IC3 industry partners.

References

[1] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS*, 2012.

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

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

[4] 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.

[5] E. M. Kornaropoulos, C. Papamanthou, and R. Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *IEEE S&P*, 2020.

[6] Paul Grubbs, Anurag Khandelwal, Marie-Sarah Lacharité, Lloyd Brown, Lucy Li, Rachit Agarwal, and Thomas Ristenpart. Pancake: Frequency smoothing for encrypted data stores. In *USENIX Security*, 2020.

[7] Care Cloud. 5 advantages of a cloud-based EHR. https://www.carecloud.com/continuum/5-advantages-of-a-cloud-based-ehr-for-small-practices/.

[8] Alex Mu-Hsing Kuo. Opportunities and challenges of cloud computing to improve health care services. *JMIR*, 2011.

[9] Microsoft. Healthcare-europe. https://www.microsoft.com/en-ie/lcc_cloud/healthcare-europe.

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

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

[12] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A parallel oblivious file system. In *CCS*, 2012.

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

[14] Jacob R. Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *FAST*, 2013.

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

[16] 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.

[17] Anrin Chakraborti and Radu Sion. ConcurORAM: High-throughput stateless parallel multi-client ORAM. In *NDSS*, 2019.

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

[19] Marie-Sarah Lacharite and Kenneth G. Paterson. Frequency-smoothing encryption: preventing snapshot attacks on deterministically encrypted data. *IACR Transactions on Symmetric Cryptology*, 2018.

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

[21] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *CRYPTO*, 2018.

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

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

[24] Kasper Green Larsen, Mark Simkin, and Kevin Yeo. Lower bounds for multi-server oblivious rams. In *TCC*, 2020.

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

[26] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *SOSP*, 2021.

[27] Securing cloud services for health. https://www.enisa.europa.eu/news/enisa-news/securing-cloud-services-for-health.

[28] French decision to have microsoft host health data hub still attracts criticism. https://www.euractiv.com/section/health-consumers/news/french-decision-to-have-microsoft-host-health-data-hub-still-attracts-criticism/.

[29] Microsoft cloud services will store and process eu data within the eu. https://www.privacy-ticker.com/microsoft-cloud-services-will-store-and-process-eu-data-within-the-eu/.

[30] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *SOSP*, 2011.

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

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

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

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

[35] Skyhigh Networks. http://www.skyhighnetworks.com.

[36] Richard D Schlichting and Fred B Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *TOCS*, 1983.

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

[38] Original buttermilk pancakes - (short stack). https://www.ihop.com/en/menu/world-famous-buttermilk-pancakes-and-crepes/original-buttermilk-pancakes-short-stack.

[39] Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, 2004.

[40] Scott Lystig Fritchie. Chain replication in theory and in practice. In *Erlang*, 2010.

[41] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, 2010.

[42] Apache zookeeper. https://zookeeper.apache.org/.

[43] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. 1987.

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

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

[46] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram and applications. In *TCC*, 2016.

[47] T-H Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel ram. In *TCC*, 2018.

[48] T-H Hubert Chan and Elaine Shi. Circuit opram: Unifying statistically and computationally secure orams and oprams. In *TCC*, 2017.

[49] Gareth T Davies, Christian Janson, and Daniel P Martin. Client-oblivious opram. In *ICICS*, 2020.

A Security Proofs

Model Assumptions. In all following cases, we assume that the adversary has access to SHORTSTACK's encryption of the KV store KV', and a transcript τ generated by SHORTSTACK in response to q queries drawn from a possibly time varying distribution. We assume network latencies between L1, L2 and L3 proxy servers to be stable, with variance indistinguishable from noise. We do not hide the timing of client accesses.

For the case of static distributions, we let the adversary select up to f proxy servers that are failed. We also note that the failure repair (i.e., detecting and eliminating the failed node, and ensuring the remaining nodes can resume operation) in chain replication occurs in bounded time T_{repair} – small enough that there is no *detectable* dip in the throughput. This is consistent with our failure model described in §4.3 and confirmed in our experiments in §6.2.

For the case of dynamic distributions, in addition to the assumptions stated for static distributions, we assume SHORTSTACK L1 leader can detect distribution changes instantaneously (as in [6]). Since SHORTSTACK's distribution change detection is the same as in PANCAKE [6], and occurs at a single leader L1 server that observes keys of all client queries (§4.2, §4.4), the detection is just as accurate as in PANCAKE. Further, we assume the distribution change is propagated to all proxy servers instantaneously. If detection and/or propagation takes finite time, SHORTSTACK— as well as PANCAKE— experiences a short period of bias in distribution proportional to the difference between $\hat{\pi}$ and $\hat{\pi}'$. However, prior work [6] shows that exploiting this bias to learn anything meaningful about the underlying distribution is challenging.

We now provide some technical preliminaries, then prove SHORTSTACK security theorems.

Preliminaries. Let $\mathbf{Adv}_F^{\text{prf}}(\mathbb{B})$ be the advantage $\mathbb{B}$ has in distinguishing the pseudo-random function (PRF) F from a random oracle. Let E be an *authenticated encryption with associated data* (AEAD) scheme characterized by the triple of algorithms: (KeyGen, Enc, Dec). We define $\mathbf{Adv}_E^{\text{ror}}(\mathbb{C})$ to be the adversary's advantage in distinguishing $E.\mathsf{Enc}$ from a random oracle. Next, we define $\mathbf{Adv}_{q,\pi,\hat{\pi}}^{\text{dist}}(\mathbb{D})$ as the advantage an adversary $\mathbb{D}$ has in computationally distinguishing between the actual and estimated distributions π and $\hat{\pi}$ using q samples.

Security analysis for static distributions. We define the chance of success of an adversary $\mathbb{A}$ in attacking an encrypted

```
$\text{IND-CDFA}^{\mathbb{A}}_{b,q,S,f,\pi_0,\hat{\pi}_0,\pi_1,\hat{\pi}_1}$:
$\mathsf{KV}, \mathcal{T}, \mathsf{st}_A \leftarrow\!\!\$\ \mathbb{A}_1(f,S)$
$(\mathsf{KV}', \mathcal{C}, \delta) \leftarrow\!\!\$\ \mathsf{Init}(\hat{\pi}_b, \mathsf{KV}, S, f)$
For $i$ in 1 to $q$:
    $w \leftarrow\!\!\$\ \pi_b$
    $W \leftarrow W \cup \{w\}$
$\tau_1, \tau_2, \ldots \leftarrow \mathsf{Process}(W, \mathcal{C}, \mathcal{T}, \mathsf{KV}', \delta)$
$b' \leftarrow\!\!\$\ \mathbb{A}_3(\mathsf{st}_A, \mathsf{KV}', \tau_1, \tau_2, \ldots)$
return $b'$
```

(a) IND-CDFA

```
$\text{IND-CDDFA}^{\mathbb{A}}_{b,q,S,f,\pi_0,\hat{\pi}_0,\pi'_0,\hat{\pi}'_0,\pi_1,\hat{\pi}_1,\pi'_1,\hat{\pi}'_1}$:
$\mathsf{KV}, c, \mathcal{T}, \mathsf{st}_A \leftarrow\!\!\$\ \mathbb{A}_1(f,S)$
$(\mathsf{KV}', \mathcal{C}, \delta) \leftarrow\!\!\$\ \mathsf{Init}(\hat{\pi}_b, \mathsf{KV}, S, f)$
For $i$ in 1 to $c$:
    $w \leftarrow\!\!\$\ \pi_b$
    $W \leftarrow W \cup \{w\}$
$\tau_1, \tau_2, \ldots \leftarrow \mathsf{Process}(W, \mathcal{C}, X, \mathsf{KV}', \delta)$
For $i$ in $c$ to $q$:
    $w \leftarrow\!\!\$\ \pi'_b$
    $W' \leftarrow W' \cup \{w\}$
$\tau'_1, \tau'_2, \ldots \leftarrow \mathsf{Process}(W', \mathcal{C}, X, \mathsf{KV}', \delta, \hat{\pi}'_b)$
$b' \leftarrow\!\!\$\ \mathbb{A}_3(\mathsf{st}_A, \mathsf{KV}', \tau_1, \tau'_1, \tau_2, \tau'_2, \ldots)$
return $b'$
```

(b) IND-CDDFA

Figure 15: **Security games in SHORTSTACK.**

KV store EKV via frequency analysis, as the advantage it has in guessing the right underlying distribution in the game IND-CDFA (Figure 15(a)) as follows:

$$\mathbf{Adv}^{\text{ind-cdfa}}_{\mathsf{EKV}}(\mathbb{A}) = \left|\Pr[\text{IND-CDFA}^{\mathbb{A}}_{1,q,S,f} \Rightarrow 1] - \Pr[\text{IND-CDFA}^{\mathbb{A}}_{0,q,S,f} \Rightarrow 1]\right| .$$

We already discussed the game description in §5 of the main paper — we now focus on detailing the simulators employed in SHORTSTACK to translate the distributed, fault-tolerant query execution in SHORTSTACK to a sequentialized version compatible with the IND-CDFA game.

```
$\mathsf{Process}(W, \mathcal{C}, \mathcal{T}, \mathsf{KV}', \delta)$:
$C_{\mathsf{L1}}, C_{\mathsf{L2}}, C_{\mathsf{L3}} \leftarrow \mathcal{C}$
$k_F \leftarrow\!\!\$\ \mathcal{K}; k_{AE} \leftarrow\!\!\$\ \mathcal{K}$;
For $(k,v)$ in $W$:
    $s_{\mathsf{L1}} \leftarrow\!\!\$\ C_{\mathsf{L1}}$
    $s_{\mathsf{L1}}.\mathrm{ProcessQuery}(k,v)$
For $s_{\mathsf{L2}}$ in $C_{\mathsf{L2}}$:
    $\mathsf{Interleave}(s_{\mathsf{L2}}.\mathrm{Queue})$
    While $s_{\mathsf{L2}}.\mathrm{Queue}$ is non-empty:
        $s_{\mathsf{L2}}.\mathrm{Process}()$
For $s_{\mathsf{L3}}$ in $C_{\mathsf{L3}}$:
    $\mathsf{Interleave}(s_{\mathsf{L3}}.\mathrm{Queue})$
    While $s_{\mathsf{L3}}.\mathrm{Queue}$ is non-empty:
        $(k',v) \leftarrow\!\!\$\ s_{\mathsf{L3}}.\mathrm{Process}(\delta)$
        $\beta_{s_{\mathsf{L3}}} \leftarrow \beta_{s_{\mathsf{L3}}} \cup (k',v)$
$\tau_1, \tau_2, \ldots, \tau_{|C_{\mathsf{L3}}|} \leftarrow\!\!\$\ \mathsf{Transform}(\mathcal{T}, C_{\mathsf{L3}}, \beta_1, \beta_2, \ldots, \beta_{|C_{\mathsf{L3}}|})$
return $\tau_1, \tau_2, \ldots, \tau_{|C_{\mathsf{L3}}|}$
```

Figure 16: SHORTSTACK Process algorithm.

Simulating SHORTSTACK *query processing logic:* Figure 16 shows the Process function for SHORTSTACK; note that the function *simulates* the behavior of SHORTSTACK on the query vector W.

It works by first generating the intermediate transcripts β assuming no failures, going layer by layer and executing processing logic at appropriate servers in SHORTSTACK layers (L1, L2, L3). Specifically, in the first loop, queries are first processed at a randomly chosen chain s_{L1} in C_{L1}. In the second loop, for each chain s_{L2} in L2 layer, the enqueued queries are first shuffled using the Interleave function to simulate the effect of network-induced inter-leavings between servers in L1 layer and s_{L2}, and then processed according to L2 processing logic. Note that the shuffling done by the Interleave function does not depend on the contents of the queries. Similarly, in the third loop, for each server s_{L3} in L3 layer, queries are first shuffled using the Interleave function to simulate network reorderings, and then processed according to L3 Process logic. The generated queries are subsequently added to the transcript $\beta_{s_{\mathsf{L3}}}$. Finally, Process then uses a Transform simulator to simulate the effect of failures and generate the final transcripts τ from β, as we describe next.

```
$\mathsf{Transform}(X, H, \beta_1, \beta_2, \ldots, \beta_{|C_{\mathsf{L3}}|})$:
If $\mathcal{T}.\mathsf{Empty}()$:
    return $\beta_1, \beta_2, \ldots, \beta_{|C_{\mathsf{L3}}|}$
Else:
    Do:
        $(n,t,\gamma,r) \leftarrow \mathcal{T}.\mathsf{Pop}()$
    While $n \notin C_{\mathsf{L3}}$;
    $H \leftarrow H \setminus \{n\}$
    For $(k,v)$ in $\beta_n^{t-\gamma,\infty}$:
        $\rho[\mathcal{H}_H(k)] \leftarrow \rho[\mathcal{H}_H(k)] \cup \{(k,v)\}$
    $\beta_n \leftarrow \beta_n^{0,t}$
    For $s$ in $H$:
        $\rho[s] \leftarrow \mathsf{Shuffle}(\rho[s])$
        $\beta_s \leftarrow \beta_s^{0,t+r} \cup \mathsf{Interleave}(\rho[s] \cup \beta_s^{t+r,\infty})$
    return $\mathsf{Transform}(\mathcal{T}, H, \beta_1, \beta_2, \ldots, \beta_{|C_{\mathsf{L3}}|})$
```

Figure 17: SHORTSTACK Transform simulator for failures.

Failure simulation using Transform*:* As noted in §4.3 of the main paper, SHORTSTACK handles the failures in layers L1 and L2 transparently using chain replication. Also, recall that failure repairs in SHORTSTACK (i.e., detecting and eliminating the failed node, and ensuring the remaining nodes can resume operation) occurs in bounded time which is small enough to render failures at the L1 and L2 layers undetectable to an adversary (consistent with our failure model in §4.3 and empirical validation in §6.2), leaving only the handling of L3 failures. We model this using the Transform simulator — a recursive algorithm that repeatedly applies the effect of L3 server failure events in $\mathcal{T}$ on the intermediate transcripts β, in the order that the failures occur. We detail the operation of Transform next; note that we use the term $\beta_n^{t_1,t_2}$ to denote the subset of queries from the intermediate transcript β enqueued at server n in the time interval $[t_1, t_2]$.

In the base case with no L3 server failures (i.e., $\mathcal{T}$ is empty), the output transcript τ is the same as the input transcript β. Otherwise, the algorithm dequeues failure events from the queue $\mathcal{T}$, ignoring non-L3 failures. When an L3 failure event $e = (n,t,\gamma,r)$ is encountered, Transform takes the following actions to simulate its behavior. First, n is removed from the set of healthy servers H at the L3 layer. Since $t - \gamma$ is the time at which the last query was acknowledged at server n prior to failure, $\beta_n^{t-\gamma,\infty}$ denotes all the queries that were either unacknowledged and in-flight to the server, or unsent queries after the failure occurred at n. The second step taken by Transform for simulating a failure event is therefore to repartitions all queries in $\beta_n^{t-\gamma,\infty}$ across the remaining healthy servers using the consistent hash function $\mathcal{H}_H$, storing the repartitioned queries in ρ. Third, the server n's output transcript is truncated to $\beta_n^{0,t}$, the set of all the queries from the beginning of Process simulation to the time t of the failure. Next, the repartitioned queries ρ are appended to the transcripts of the corresponding healthy servers, after randomly shuffling them, and interleaving with the remaining queries $\beta_s^{t+r,\infty}$ (where r is the failure recovery time) at the healthy server to simulate network reorderings. The random shuffle is necessary for the secure replay of queries on L3 failure, as described in §4.3 of the main paper. It makes the order of the replayed queries independent of the original queries, hiding any correlation which the adversary could otherwise exploit to learn about the sharding scheme at L2 layer and thus learning distribution sensitive information. Finally, the Transform function enters recursion with the updated transcripts and set of healthy servers to process the remaining set of failure events.

Security guarantees due to $\mathcal{P}$: The underlying oblivious data access scheme, PANCAKE [6] ($\mathcal{P}$) used in SHORTSTACK provides the following security guarantee, which we leverage to prove SHORTSTACK security: an adversary cannot distinguish the output transcript generated by repeated application of $\mathcal{P}$.Batch() on a sequence of q queries, from a sequence of $q \cdot B$ uniformly distributed accesses to random bit strings, given that the PRF and encryption functions are replaced by truly random functions and $\mathcal{P}$'s estimate of the underlying distributions ($\hat{\pi}$) is the same as the distribution (π) from which queries are sampled. Less formally, the output transcript generated by $\mathcal{P}$ is independent of the underlying access distribution π. This guarantee is provided by PANCAKE.

We are now ready to prove SHORTSTACK's security under IND-CDFA.

Theorem 2 (IND-CDFA Security)**.** *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi_0, \hat{\pi}_0, \pi_1, \hat{\pi}_1$ be query distributions. For any q-query* IND-CDFA *adversary $\mathbb{A}$ against* SHORTSTACK *there exist adversaries $\mathbb{B}$, $\mathbb{C}$, $\mathbb{D}_1$, $\mathbb{D}_2$ such that*

$$\mathbf{Adv}^{\textit{ind-cdfa}}_{\text{SHORTSTACK}}(\mathbb{A}) \leq \mathbf{Adv}^{\textit{prf}}_{F}(\mathbb{B}) + \mathbf{Adv}^{\textit{ror}}_{E}(\mathbb{C})$$
$$+ \mathbf{Adv}^{\textit{dist}}_{Q,\pi_0,\hat{\pi}_0}(\mathbb{D}_1) + \mathbf{Adv}^{\textit{dist}}_{Q,\pi_1,\hat{\pi}_1}(\mathbb{D}_2)$$

where F and E are the PRF and AE schemes used by SHORTSTACK*. Adversaries $\mathbb{B}, \mathbb{C}, \mathbb{D}_1, \mathbb{D}_2$, each run in same time as $\mathbb{A}$, and make Q queries.*

Proof. We prove the above theorem using a sequence of standard cryptographic game transitions and reductions. We first substitute Init and Process in game IND-CDFA with the algorithms in SHORTSTACK (see Figure 8 of the main paper and Figure 16). Next, we replace the PRF F and the encryption scheme E with a truly random function and replace $\hat{\pi}_0$ and $\hat{\pi}_1$ with π_0 and π_1 respectively to obtain game G. Clearly, the difference between the success of an adversary $\mathbb{A}$ in the game G and IND-CDFA can be upper bounded by the sum of advantages of (i) the PRF adversary $\mathbb{B}$ against the PRF scheme F, (ii) the real-or-random adversary $\mathbb{C}$ against the encryption scheme E, (iii) the distribution adversary $\mathbb{D}_1$ in computationally distinguishing between $\hat{\pi}_0$ and π_0, and, (iv) the distribution adversary $\mathbb{D}_2$ in computationally distinguishing between $\hat{\pi}_1$ and π_1:

$$\left| \Pr_{b \sim \{0,1\}}[\text{IND-CDFA}^{\mathbb{A}}_{b,q,S,f} \Rightarrow b] - \Pr[G^{\mathbb{A}}_{b,q,S,f} \Rightarrow b] \right| \leq$$
$$\mathbf{Adv}^{\text{prf}}_{F}(\mathbb{B}) + \mathbf{Adv}^{\text{ror}}_{E}(\mathbb{C}) + \mathbf{Adv}^{\text{dist}}_{Q,\pi_0,\hat{\pi}_0}(\mathbb{D}_1) + \mathbf{Adv}^{\text{dist}}_{Q,\pi_1,\hat{\pi}_1}(\mathbb{D}_2)$$

We now argue that the underlying distribution (determined by bit b) is indistinguishable to the adversary under game G. Formally,

$$\Pr[G^{\mathbb{A}}_{0,q,S,f} \Rightarrow 1] = \Pr[G^{\mathbb{A}}_{1,q,S,f} \Rightarrow 1] \qquad (1)$$

We first make the important observation that the output of the Transform function $\tau_1, \tau_2 \ldots$ is independent of the underlying distribution if the input transcripts $\beta_1, \beta_2 \ldots$ are independent of the underlying distribution. To see why this property holds, note that the only other inputs to the function are the set of failure events $\mathcal{T}$, and the set of healthy servers H both of which are assumed to be independent of the underlying distribution. Therefore, to show Eq. 1 holds, it suffices to show that $\beta_1, \beta_2, \ldots, \beta_{|C_{\text{L3}}|}$ are independent of the underlying distribution.

The labels and values in G are random strings for both $b = 0$ and $b = 1$. Since, each access is independent, it is sufficient to prove that the label in a given access is independent of the underlying distribution π. For a particular query in β_i, let ζ_i be the random variable denoting the replica being accessed. We prove that for any replica (k, j) handled by s_i in L3 layer:

$$\Pr[\zeta_i = (k, j)] = \frac{1}{n'_i} \,.$$

where n'_i is the total number of labels handled by server s_i in L3 layer in case of no failures. Note that n'_i is independent of the underlying distribution, since the number of labels handled by an L3 server depends only on hash partitioning of ciphertext labels across L3 servers.

To see why the above equation holds, let L be the random variable representing the proxy in L1 layer that receives the query from the client and forwards it to the appropriate server in L2 layer. Then,

$$\begin{aligned} \Pr[\zeta_i = (k, j)] &= \sum_{l \in S_{\text{L1}}} \Pr[\zeta_i = (k, j) \cap L = l] \\ &= \sum_{l \in S_{\text{L1}}} \Pr[\zeta_i = (k, j) | L = l] \cdot \Pr[L = l] \end{aligned}$$

Since client queries are *randomly* load balanced across L1 proxy servers, $P[L=l] = 1/|S_{\text{L1}}|$. Therefore,

$$P[\zeta_i = (k,j)] = \sum_{l \in S_{\text{L1}}} \frac{\Pr[\zeta_i = (k,j)|L=l]}{|S_{\text{L1}}|}$$

Moreover, since each L1 proxy server independently applies oblivious data access scheme $\mathcal{P}$, we have:

$$\Pr[\zeta_i = (k,j)|L=l] = \frac{1}{n'_i}$$

and therefore,

$$P[\zeta_i = (k,j)] = \sum_{l \in S_{\text{L1}}} \frac{1}{n'_i * |S_{\text{L1}}|} = \frac{1}{n'_i}$$

completing the proof. □

Security analysis for dynamic distributions. We introduce a new game, Indistinguishability under Chosen Dynamic Distribution and Failure Attack or IND-CDDFA (Figure 15(b)) for reasoning about the security of SHORTSTACK under dynamic distributions; it is a generalization of IND-CDFA with two major changes:

- The adversary additionally outputs a query index c at which the distribution changes.
- The distributed proxy scheme's Process function takes an additional argument for the new distribution, π'_b. In SHORTSTACK, we use π'_b to recalculate the weights δ, and perform replica swapping at L1 (§4.4 in the main paper).

Security guarantees due to $\mathcal{P}$: We rely on the guarantees provided by the oblivious data access scheme, PANCAKE [6] ($\mathcal{P}$) to prove the security guarantees for SHORTSTACK. Specifically, the oblivious data access scheme $\mathcal{P}$ guarantees that no distribution sensitive information is revealed during and after the transition to the new distribution $\hat{\pi}'$, using the replica swapping approach. As with the static distribution setting, this guarantee is provided by the oblivious data access scheme used in our implementation [6].

We define the probability of success for an adversary $\mathbb{A}$ in attacking EKV as its advantage in guessing the right underlying distributions in the game IND-CDDFA:

$$\begin{aligned}\mathbf{Adv}^{\text{ind-cddfa}}_{\text{EKV}}[(\mathbb{A})] = &|\Pr[\text{IND-CDDFA}^{\mathbb{A}}_{1,q,S,f} \Rightarrow 1] - \\ &\Pr[\text{IND-CDDFA}^{\mathbb{A}}_{0,q,S,f} \Rightarrow 1]|.\end{aligned}$$

The following theorem establishes the security of SHORTSTACK under IND-CDDFA:

Theorem 3 (IND-CDDFA Security). *Let $q \geq 0$ and $Q = q \cdot B$. Let $\pi_0, \hat{\pi}_0, \pi'_0, \hat{\pi}'_0, \pi_1, \hat{\pi}_1, \pi'_1, \hat{\pi}'_1$ be query distributions. For any q-query* IND-CDDFA *adversary $\mathbb{A}$ against* SHORTSTACK *there exist adversaries $\mathbb{B}$, $\mathbb{C}$, $\mathbb{D}_1$, $\mathbb{D}_2$, $\mathbb{D}_3$,$\mathbb{D}_4$ such that*

$$\begin{aligned}\mathbf{Adv}^{\textit{ind-cddfa}}_{\text{SHORTSTACK}}[(\mathbb{A})] \leq \mathbf{Adv}^{\textit{prf}}_{F}[(\mathbb{B})] + \mathbf{Adv}^{\textit{ror}}_{E}[(\mathbb{C})] \\ + \mathbf{Adv}^{\textit{dist}}_{Q,\pi_0,\hat{\pi}_0}[(\mathbb{D}_1)] + \mathbf{Adv}^{\textit{dist}}_{Q,\pi'_0,\hat{\pi}'_0}[(\mathbb{D}_2)] \\ + \mathbf{Adv}^{\textit{dist}}_{Q,\pi_1,\hat{\pi}_1}[(\mathbb{D}_3)] + \mathbf{Adv}^{\textit{dist}}_{Q,\pi'_1,\hat{\pi}'_1}[(\mathbb{D}_4)]\end{aligned}$$

where F and E are the PRF and AE schemes used by SHORTSTACK. *Adversaries $\mathbb{B},\mathbb{C},\mathbb{D}_1,\mathbb{D}_2,\mathbb{D}_3,\mathbb{D}_4$ each run in same time as $\mathbb{A}$, and make Q queries.*

Proof. We first give a proof sketch, followed by the details. Our proof is similar to that for IND-CDFA security — we show that the transcripts generated by SHORTSTACK both before and after the distribution change are independent of the underlying distributions. Clearly, IND-CDFA security already guarantees that the output transcripts before the distribution changes and after replica swapping phase completes in SHORTSTACK are independent of the underlying distribution. The interim period comprises detecting the distribution change and performing replica swaps in response. Our distribution change detection occurs at a single leader L1 proxy server that observes keys of all client queries (§4.2, §4.4 in the main paper), and employs the same detection mechanism as PANCAKE. As such, the scheme ensures that the output transcript is independent of the underlying distribution in the interim phase, provided the distribution change occurs instantaneously. If the detection takes finite time, SHORTSTACK— as well as PANCAKE— experience a short period of bias in distribution proportional to the difference between $\hat{\pi}$ and $\hat{\pi}'$. However, prior work [6] shows that exploiting this bias to learn anything meaningful about the underlying distribution is challenging. We now detail our formal proof, leveraging the security of PANCAKE to ensure output transcripts are independent of the access distributions π, π'.

Proof Details. Similar to the proof for IND-CDFA security, we substitute Init and Process in IND-CDDFA game with the algorithms in SHORTSTACK. Next, we replace the PRF F and the encryption scheme E with a truly random function, and replace $\hat{\pi}_0, \hat{\pi}'_0, \hat{\pi}_1$ and $\hat{\pi}'_1$ with π_0, π'_0, π_1 and π'_1 respectively to obtain game G. Again, the difference between the success of an adversary $\mathbb{A}$ in the game G and IND-CDFA can be upper bounded by the sum of advantages of (i) the PRF adversary $\mathbb{B}$ against the PRF scheme F, (ii) the real-or-random adversary $\mathbb{C}$ against the encryption scheme E, (iii) the distribution adversaries $\mathbb{D}_1, \mathbb{D}_2, \mathbb{D}_3$ and $\mathbb{D}_4$ in computationally distinguishing between distributions $\hat{\pi}_0 \leftrightarrow \pi_0$, $\hat{\pi}'_0 \leftrightarrow \pi'_0$, $\hat{\pi}_1 \leftrightarrow \pi_1$, and

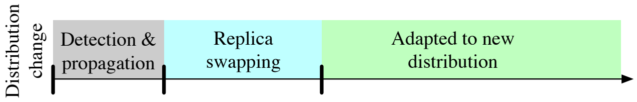


Figure 18: **Three distinct phases in SHORTSTACK after distribution change.** See proof of IND-CDDFA security for details.

$\hat{\pi}'_1 \leftrightarrow \pi'_1$, respectively:

$$\Big|\Pr_{b\sim\{0,1\}}[\text{IND-CDDFA}^{\mathbb{A}}_{b,q,S,f} \Rightarrow b] - \Pr[G^{\mathbb{A}}_{b,q,S,f} \Rightarrow b]\Big| \le \mathbf{Adv}^{\text{prf}}_{F}[(\mathbb{B})] + \mathbf{Adv}^{\text{ror}}_{E}[(\mathbb{C})] + \mathbf{Adv}^{\text{dist}}_{Q,\pi_0,\hat{\pi}_0}[(\mathbb{D}_1)] + \mathbf{Adv}^{\text{dist}}_{Q,\pi'_0,\hat{\pi}'_0}[(\mathbb{D}_2)] + \mathbf{Adv}^{\text{dist}}_{Q,\pi_1,\hat{\pi}_1}[(\mathbb{D}_3)] + \mathbf{Adv}^{\text{dist}}_{Q,\pi'_1,\hat{\pi}'_1}[(\mathbb{D}_4)] \qquad (2)$$

We now argue that the underlying distribution is indistinguishable to the adversary. Formally,

$$\Pr_{b\sim\{0,1\}}[G^{\mathbb{A}}_{b,q,S,f} \Rightarrow b] = 1/2 \qquad (3)$$

In other words,

$$\Pr[G^{\mathbb{A}}_{0,q,S,f} \Rightarrow 1] = \Pr[G^{\mathbb{A}}_{1,q,S,f} \Rightarrow 1]$$

$\tau_1, \tau_2, \ldots, \tau_{|C_{\text{L3}}|}$ are already independent of the underlying distribution π due to IND-CDFA security. For $\tau'_1, \tau'_2, \ldots, \tau'_{|C_{\text{L3}}|}$ transcripts, we consider three distinct phases post distribution change, as shown in Figure 18: detection and propagation, replica swapping, and post-adaptation. For the subset of transcripts $\tau'_1, \tau'_2, \ldots, \tau'_{|C_{\text{L3}}|}$ that lie in post-adaptation phase, independence from the underlying distribution also follows from IND-CDFA security. For subset of transcripts $\tau'_1, \tau'_2, \ldots, \tau'_{|C_{\text{L3}}|}$ that lie in the detection and propagation and replica swapping phases, we leverage the security provided by the oblivious data access scheme $\mathcal{P}$, and the distribution change atomicity afforded by the atomic replica initiation protocol. In particular, $\mathcal{P}$ ensures that as long as the distribution change is atomic and instantaneous (as discussed above), the output transcript during the detection and propagation and replica swapping phases is independent of the underlying distribution. At the same time, distribution change atomicity ensures the existence of an instant of time t_{change} when SHORTSTACK switches from processing queries according to $\hat{\pi}$ to processing them according to $\hat{\pi}'$ (§4.4 of the main paper). Combining the above arguments shows that the transcripts $\tau'_1, \tau'_2, \ldots, \tau'_{|C_{\text{L3}}|}$ are indeed independent of the underlying distribution π'. Therefore, an adversary cannot use the output transcripts $\tau_1, \tau_2, \ldots, \tau_{|C_{\text{L3}}|}$ or $\tau'_1, \tau'_2, \ldots, \tau'_{|C_{\text{L3}}|}$ to distinguish between two input distributions to the IND-CDDFA game, completing our proof. □

Messages:

`prepare`, `prepareACK`: Marks beginning and end of prepare phase
`commit`, `commitACK`: Marks beginning and end of commit phase
`flushEOF`: Acknowledges all pending queries are flushed

L1 Leader:

On distribution change:
Send `prepare` to all L1, L2, L3
Wait for all `prepareACK`s

On receiving all `prepareACK`s:
Send `commit` to all L1, L2, L3
Wait for all `commitACK`s

On receiving all `commitACK`s:
End

L1 server:

On `prepare`:
Start queueing client queries
Flush pending queries to L2
Send `flushEOF` to all L2
Send `prepareACK` to leader

On `commit`:
Stop queueing client queries
Send `commitACK` to leader

L2 server:

On `prepare`:
Wait for `flushEOF` from all L1
Flush pending queries to L3
Send `flushEOF` to all L3
Send `prepareACK` to leader

On `commit`:
Send `commitACK` to leader

L3 server:

On `prepare`:
Wait for `flushEOF` from all L2
Flush pending queries to KV′
Send `prepareACK` to leader

On `commit`:
Send `commitACK` to leader

Figure 19: **Atomic initiation protocol for replica-swapping.**

B Distribution Change Atomicity Proof

We now prove that our 2-PC inspired protocol for atomic initiation of replica swapping guarantees the Distribution Change Atomicity Invariant. In order to prove this, we first precisely specify the details of our protocol. Note that the termination phase after replica swapping employs a similar approach. The protocol is depicted in Figure 19:

- On detecting a distribution change from $\hat{\pi}$ to $\hat{\pi}'$, the leader sends `prepare` messages to all the L1, L2 and L3 servers.
- On receiving a `prepare` message, L1 servers temporarily stop processing new queries and start queueing them up — these correspond to the new distribution $\hat{\pi}'$. They subsequently flush pending queries from $\hat{\pi}$ that have not yet been sent to L2 proxy servers. Once all such queries are flushed, they send a `flushEOF` message to all the L2 proxy servers, and a `prepareACK` message to the leader.
- L2 servers, on receiving a `prepare` message, first wait till they have received `flushEOF` from *all* L1 proxy servers, and then flush all pending queries to the L3 servers. Similar to L1 servers, they then send `flushEOF` to all the L3 servers and respond to the L1 leader with a `prepareACK` message.
- L3 servers, upon receiving a `prepare` message, take actions similar to L2 servers: wait for all `flushEOF` messages L2 servers, flush all pending queries to the KV store, and respond to the L1 leader with a `prepareACK` message.
- Once the L1 leader receives `prepareACK` from all other servers, it sends a `commit` message to all of them.
- On receiving `commit`, the L1 servers stop queueing client queries and start processing them again according to $\hat{\pi}'$, as per the replica-swapping approach employed by noise-injection schemes.

Similar to replica swapping in the single-proxy approach, for every key that gains a replica, another key must lose one. The leader proxy determines the pairs of replicas that need to be swapped, the temporary fake access distribution and other relevant information. SHORTSTACK communicates this information to the other `L1` and `L2` proxy servers by piggybacking it onto `prepare` messages. More precisely, the `L1` leader proxy computes and embeds the following information in `prepare` messages:

- The new real distribution, new fake distribution, and temporary fake distribution used during replica swapping.
- The list of keys that are to gain and lose replicas, along with the mapping between them.
- The updated mapping from plaintext keys to their replicas.

Since the protocol for initiating and terminating the replica-swapping phase takes finite time, an adversary may be able to detect that a distribution change has occurred. However, we show in §5 of the main paper that the atomicity of replica-swapping initiation and termination, coupled with the security of replica swapping in noise-injection schemes, ensures that the adversary cannot glean any information about $\hat{\pi}$ or $\hat{\pi}'$ under SHORTSTACK.

We now prove that the above protocol guarantees the distribution change atomicity invariant:

Invariant 3 (Distribution change atomicity)**.** *Once any `L3` proxy server issues a query according to $\hat{\pi}'$, all subsequent queries issued by any `L3` server must be according to $\hat{\pi}'$.*

Proof. We prove that the invariant holds for a single distribution change event from $\hat{\pi}$ to $\hat{\pi}'$. It can easily be generalized to an arbitrary sequence of distribution change events using induction.

To prove this we show that there exists an instant of time t_{change} in the protocol's execution, such that: (1) before t_{change}, all queries are processed according to the distribution $\hat{\pi}$, and (2) after t_{change}, all queries are processed using $\hat{\pi}'$. We show that $t^* =$ the time instant when the `L1` leader receives `prepareACK` from all other servers, satisfies this property.

First, we prove that a query processed according to $\hat{\pi}'$ can only be forwarded by an `L3` server after t^*. The `L1` servers start processing queries according to $\hat{\pi}'$ only after they receive a `commit` message. Such a `commit` message can only be delivered once it has been sent by the `L1` leader. A `commit` can only be sent by the `L1` leader after t^*. Hence, an `L1` server can only start processing queries according to $\hat{\pi}'$ after t^*. A query can be issued by an `L3` server only after it has been processed at an `L1` server. Hence, a query processed according to $\hat{\pi}'$ can only be forwarded by an `L3` server after t^*.

Second, we prove that queries according to $\hat{\pi}$ cannot be forwarded by an `L3` server on or after t^*. Before t^* (when `L1` leader has received all `prepareACKs`), all of the `L3` servers would have flushed any outstanding queries, before which all `L2` servers would have flushed their outstanding queries, before which all `L3` servers would have flushed their outstanding queries. Hence, it is guaranteed that by t^* all queries processed according to $\hat{\pi}$ have been issued by the `L3` servers.

Combining the above two facts implies that all queries issued by `L3` servers after t^* are those processed according to $\hat{\pi}'$, while all queries before t^* are those processed according to $\hat{\pi}$, hence proving the invariant. □