Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage

Emma Dauterman*
UC Berkeley

Vivian Fang*
UC Berkeley

Ioannis Demertzis
UC Berkeley and UC Santa Cruz

Natacha Crooks
UC Berkeley

Raluca Ada Popa
UC Berkeley

Abstract

Existing oblivious storage systems provide strong security by hiding access patterns, but do not scale to sustain high throughput as they rely on a central point of coordination. To overcome this scalability bottleneck, we present Snoopy, an object store that is both oblivious *and* scalable such that adding more machines increases system throughput. Snoopy contributes techniques tailored to the high-throughput regime to securely distribute and efficiently parallelize every system component without prohibitive coordination costs. These techniques enable Snoopy to scale similarly to a plaintext storage system. Snoopy achieves 13.7× higher throughput than Obladi, a state-of-the-art oblivious storage system. Specifically, Obladi reaches a throughput of 6.7K requests/s for two million 160-byte objects and cannot scale beyond a proxy and server machine. For the same data size, Snoopy uses 18 machines to scale to 92K requests/s with average latency under 500ms.

1 Introduction

Organizations increasingly outsource sensitive data to the cloud for better convenience, cost-efficiency and availability [31, 53, 89]. Encryption cannot fully protect this data: how the user accesses data (the "access pattern") can leak sensitive information to the cloud [13, 28, 37, 48, 50, 52]. For example, the frequency with which a doctor accesses a medication database might reveal a patient's diagnosis.

Oblivious object stores allow clients to outsource data to a storage server without revealing access patterns to the storage server. A rich line of work has shown how to build efficient oblivious RAMs (ORAMs), which can be used to construct oblivious object stores [8, 14, 26, 33, 72, 82, 85, 91–93, 101]. In order to be practical for applications, oblivious storage must provide many of the same properties as plaintext storage. Prior work has shown how to reduce latency [65, 82, 93], scale to large data sizes via data parallelism [59], and improve request throughput [26, 85, 101]. Despite this progress, leveraging task parallelism to *scale for high-throughput workloads* remains an open problem: existing oblivious storage systems do not scale.

Identifying the scalability bottleneck. Scalability bottlenecks are system components that must perform computation for every request and cannot be parallelized. These bottlenecks limit the overall system throughput; once their maximum throughput has been reached, adding resources to the system no longer improves performance. To scale, plaintext object stores traditionally shard objects across servers, and clients can route their queries to the appropriate server. Unfortunately, this approach is insecure for oblivious object stores because it reveals the mapping of objects to partitions [13, 37, 48, 50, 52]. For example, if clients query different shards, the attacker learns that the requests were for different objects.

To understand why scaling oblivious storage is hard, we examine two properties oblivious storage systems traditionally satisfy. First, systems typically maintain a dynamic mapping (hidden from the untrusted server) between the logical layout and physical layout of the outsourced data. Clients must look up their logical key using the freshest mapping and remap it to a new location after every access, creating a central point of coordination. Second, for efficient access, oblivious systems typically store data in a hierarchical or tree-like structure, creating a bottleneck at the root [82, 92, 93].

Thus high-throughput oblivious storage systems are all built on hierarchical [92] or tree-like [82, 93] structures and either require a centralized coordination point (e.g., a query log [14, 101] or trusted proxy [8, 26, 85, 91]) or inter-client communication [10]. We ask: *How can we build an oblivious object store that handles high throughput by scaling in the same way as a plaintext object store?*

Removing the scalability bottleneck. In this work, we propose Snoopy (scalable nodes for oblivious object repository), a high-throughput oblivious storage system that scales

*Equal contribution.

SOSP '21, October 26–29, 2021, Virtual Event, Germany

similarly to a plaintext storage system. While our system is secure for any workload, we design it for high-throughput workloads. Specifically, we develop techniques for grouping requests into equal-sized batches for each partition regardless of the underlying request distribution and with minimal cover traffic. These techniques enable us to efficiently partition and securely distribute every system component without prohibitive coordination costs.

Like prior work, Snoopy leverages hardware enclaves for both performance and security [3, 65, 86]. Hardware enclaves makes it possible to (1) deploy the entire system in a public cloud; (2) reduce network overheads, as private and public state can be located on the same machine; and (3) support multiple clients without creating a central point of attack. This is in contrast with the traditional trusted proxy model (Figure 1), which can be both a deployment headache and a scalability concern. Hardware enclaves do not entirely solve the problem of hiding access patterns for oblivious storage: enclave side channels allow attackers to exploit data-dependent memory accesses to extract enclave secrets [12, 41, 55, 57, 67, 88, 98, 102]. To defend against these attacks, we must ensure that all algorithms running inside the enclave are oblivious, meaning that memory accesses are data-independent. Existing work targets latency-sensitive deployments [3, 65, 86] and is prohibitively expensive for the concurrent, high-throughput deployment we target. We instead leverage our oblivious partitioning scheme to design new algorithms tailored to our setting.

We experimentally show that Snoopy scales to achieve high throughput. The state-of-the-art oblivious storage system Obladi [26] reaches a throughput of 6,716 reqs/sec with average latency under 80ms for two million 160-byte objects and cannot scale beyond a proxy machine (32 cores) and server machine (16 cores). In contrast, Snoopy uses 18 4-core machines to scale to a throughput of 92K reqs/sec with average latency under 500ms for the same data size, achieving a 13.7× improvement over Obladi. We report numbers with 18 machines due to cloud quota limits, not because Snoopy stops scaling. We formally prove the security of the entire Snoopy system, independent of the request load.

1.1 Summary of techniques

Snoopy is comprised of two types of entities: *load balancers* and *subORAMs* (Figure 1). Load balancers assemble batches of requests, and subORAMs, which store data partitions, process the requests. In order to securely achieve horizontal scaling, we must consider how to design both the load balancer and subORAM to (1) leverage efficient oblivious algorithms to defend against memory-based side-channel attacks, and (2) be easy to partition without incurring coordination costs.

Challenge #1: Building an oblivious load balancer. To protect the contents of the requests, our load balancer design must guarantee that (1) the batch structure leaks no information about the requests, and (2) the process of constructing

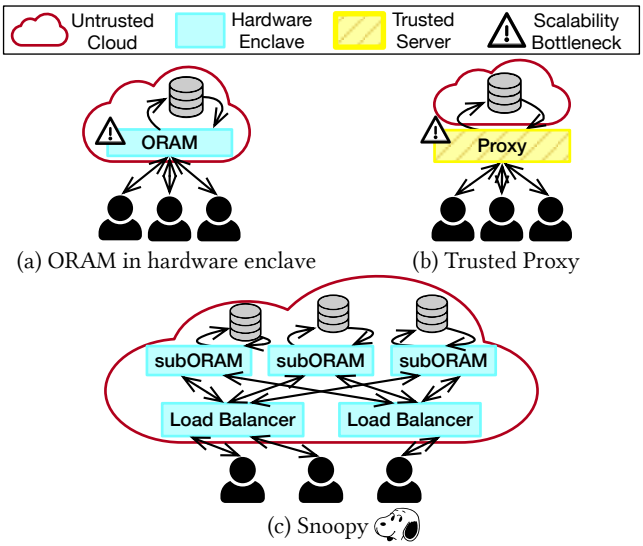


Figure 1. Different oblivious storage system architectures: (a) ORAM in a hardware enclave is bottlenecked by the single machine, (b) ORAM with a trusted proxy is bottlenecked by the proxy machine, and (c) Snoopy can continue scaling as more subORAMs and load balancers are added to the system.

these batches is oblivious and efficient. Furthermore, we need to design our oblivious algorithm such that we can add load balancers without incurring additional coordination costs.

Approach. We build an efficient, oblivious algorithm that groups requests into batches without revealing the mapping between requests and subORAMs. We size batches using only public information, ensuring that the load balancer never drops requests and the batch size does not leak information. Our load balancer design enables us to run load balancers independently and in parallel, allowing Snoopy to scale past the capacity of a single load balancer (§4).

Challenge #2: Designing a high-throughput subORAM. To ensure that Snoopy can achieve high throughput, we need a subORAM design that efficiently processes large batches of requests and defends against enclave side-channel attacks. Existing ORAMs that make use of hardware enclaves [3, 65, 86] only process requests sequentially and are a poor fit for the high-throughput scenario we target.

Approach. Rather than building batching support into an existing ORAM scheme, we design a new ORAM that only supports batched accesses. We observe that in the case where data is partitioned over many subORAMs, a single scan amortized over a large batch of requests is concretely cheaper than servicing the batch using ORAMs with polylogarithmic access costs [3, 65, 86], particularly in the hardware enclave setting. We leverage a specialized data structure to process batches efficiently and obliviously in a single linear scan (§5).

Challenge #3: Choosing the optimal configuration. The design of Snoopy makes it possible to scale the system by adding both load balancers and subORAMs. An application

developer needs to know how to configure the system to meet certain performance targets while minimizing cost.

Approach. To solve this problem, we design a planner that, given a minimum throughput, maximum average latency, and data size, outputs a configuration minimizing cost (§6).

Limitations. Snoopy is designed specifically to overcome ORAM's scalability bottleneck to support high-throughput workloads, as solutions already exist for low-throughput, low-latency workloads [82, 93]. In the low-throughput regime, although Snoopy is still secure, its latency will likely be higher than that of non-batching systems like ConcurORAM [14], TaoStore [85], or PrivateFS [101]. For large data sizes and low request volume, a system like Shroud [59] will leverage resources more efficiently. Snoopy can use a different, latency-optimized subORAM with a shorter epoch time if latency is a priority. We leave for future work the problem of adaptively switching between solutions that are optimal under different workloads.

2 Security and correctness guarantees

We consider a cloud attacker that can:

- control the entire cloud software stack outside the enclave (including the operating system),
- view (encrypted) network traffic arriving at and within the cloud (including traffic sent by clients and message timing),
- view or modify (encrypted) memory outside the enclaves in the cloud, and
- observe access patterns between the enclaves and external memory in the cloud.

We design Snoopy on top of an abstract enclave model where the attacker controls the software stack outside the enclave and can observe memory access patterns but cannot learn the contents of the data inside the processor. Snoopy can be used with any enclave implementation [9, 25, 56]; we chose to implement Snoopy on Intel SGX as it is publicly available on Microsoft Azure. Enclaves do not hide memory access patterns, enabling a large class of side-channel attacks, including but not limited to cache attacks [12, 41, 67, 88], branch prediction [57], paging-based attacks [98, 102], and memory bus snooping [55]. By using oblivious algorithms, Snoopy defends against this class of attacks. Snoopy does not defend against enclave integrity attacks such as rollback [73] and transient execution attacks [19, 78, 87, 96, 97, 99, 100], which we discuss in greater detail below.

We defend against memory access patterns to both data and code by building oblivious algorithms on top of an oblivious "compare-and-set" operator. While our source code defends against access patterns to code, we do not ensure that the final binary does, as other factors like compiler optimizations and cache replacement policies may leak information (existing solutions may be employed here [38, 58]).

Timing attacks. A cloud attacker has access to three types of timing information: (1) when client requests arrive, (2) when inter-cloud processing messages are sent/received, and (3) when client responses are sent. Snoopy allows the attacker to learn (1). In theory, these arrival times can leak data, and so we could hide when clients send requests and how many they send by requiring clients to send a constant number of requests at predefined time intervals [4]; we do not take this approach because of the substantial overhead and because, for some applications, clients may not always be online. Snoopy ensures that (2) and (3) do not leak request contents; the time to execute a batch depends entirely on public information, as defined in §2.1.

Data integrity and protection against rollback attacks. Snoopy guarantees the integrity of the stored objects in a straightforward way: for memory within the enclave, we use Intel SGX's built-in integrity tree, and for memory outside the enclave, we store a digest of each block inside the enclave. We assume that the attacker cannot roll back the state of the system [73]. We discuss how Snoopy can integrate with existing rollback-attack solutions in §9.

Attacks out of scope. We build on an abstract enclave model where the attacker's power is limited to viewing or modifying external memory and observing memory access patterns (we formalize this as an ideal functionality in §B). Any attack that breaks the abstract enclave model is out of scope and should be addressed with techniques complementary to Snoopy. For example, we do not defend against leakage due to power consumption [20, 68, 94] or denial-of-service attacks due to memory corruptions [39, 49]. We additionally consider transient execution attacks [19, 78, 87, 96, 97, 99, 100] to be out of scope; in many cases, these have been patched by the enclave vendor or the cloud provider. These attacks break Snoopy's assumptions (and hence guarantees) as they allow the attacker to, in many cases, extract enclave secrets. We note that, Snoopy's design is not tied to Intel SGX, and also applies to academic enclaves like MI6 [9], Keystone [56], or Sanctum [25], which avoid many of the drawbacks of Intel SGX.

We also do not defend against denial-of-service attacks; the attacker may refuse queries or even delete the clients' data.

Clients. For simplicity, in the rest of the paper, we describe the case where all clients are honest. We make this simplification to focus on protecting client requests from the server, a technical challenge that motivates our techniques. However, in practice, we might not want to trust every client with read and write access to every object in the system. Adding access-control lookups to our system is fairly straightforward and requires an oblivious lookup in an access-control matrix to check a client's privileges for a given object. We can perform this check obliviously via a recursive lookup in Snoopy (we describe how this works in §D). Supporting access control in Snoopy ensures that compromised clients

cannot read or write data that they do not have access to. Furthermore, if compromised clients collude with the cloud, the cloud does not learn anything beyond the public information that it already learns (specified in §2.1) and the results of read requests revealed by compromised clients.

Linearizability. Because we handle multiple simultaneous requests, we must provide some ordering guarantee. Snoopy provides linearizability [43]: if one operation happens after another in real time, then the second will always see the effects of the first (see §4.3 for how we achieve this). We include a linearizability proof in §C.

2.1 Formalizing security

We formalize our system and prove its security in Appendix B. We build our security definition on an enclave ideal functionality (representing the abstract enclave model), which provides an interface to load a program onto a network of enclaves and then execute that program on an input. Execution produces the program output, as well as a *trace* containing the network communication and memory access patterns generated as a result of execution (what the adversary has access to in the abstract enclave model).

The Snoopy protocol allows the attacker to learn public information such as the number of requests sent by each client, request timing, data size (number of objects and object size), and system configuration (number of load balancers and subORAMs); this public information is standard in oblivious storage. Snoopy protects private information, including the data content and, for each request, the identity of the requested object, the request type, and any read or write content. To prove security, we show how to simulate all accesses based solely on public information (as is standard for ORAM security [33]). Our construction is secure if an adversary cannot distinguish whether it is interacting with enclaves running the real Snoopy protocol (the "real" experiment) or an ideal functionality that interacts with enclaves running a simulator program that only has access to public information (the "ideal" experiment) from the trace generated by execution. We now informally define these experiments, delegating the formal details to Figure 18 in the appendix.

Real and ideal experiments (informal). In the real experiment, we load the protocol Π (either our Snoopy protocol or our subORAM protocol, depending on what we are proving security of) onto a network of enclaves and execute the initialization procedure (the adversary can view the resulting trace). Then, the adversary can run the batch access protocol specified by Π on any set of queries and view the trace. The adversary repeats this process a polynomial number of times before outputting a bit.

The ideal experiment proceeds in the same way as the real experiment, except that, instead of interacting with enclaves running Π, the adversary interacts with an ideal functionality that in turn interacts with the enclaves running the simulator program. The adversary can view the traces generated by the simulator enclaves. The goal of the adversary is to distinguish between these experiments. We describe both experiments more formally in Figure 18.

Using these experiments, we present our security definition:

Definition 1. The oblivious storage scheme Π is secure if for any non-uniform probabilistic polynomial-time (PPT) adversary Adv, there exists a PPT Sim such that

$$\left|\Pr\left[\mathbf{Real}_{\Pi,\mathsf{Adv}}^{\mathrm{OSTORE}}(\lambda)=1\right]-\Pr\left[\mathbf{Ideal}_{\mathsf{Sim},\mathsf{Adv}}^{\mathrm{OSTORE}}(\lambda)=1\right]\right|\leq \mathrm{negl}(\lambda)$$

where λ is the security parameter, the real and ideal experiments are defined informally above and formally in Figure 18 (see appendix), and the randomness is taken over the random bits used by the algorithms of Π, Sim, and Adv.

We prove security in a modular way, which enables future systems to make standalone use of our subORAM design. We note that our subORAM scheme is secure only if the batch received contains unique requests (this property is guaranteed by our load balancer). We describe these requirements formally and prove security in Definition 2 in the appendix. We prove the security of Snoopy using any subORAM scheme that is secure under this modified definition.

Theorem 1. *Given a two-tiered oblivious hash table [16], an oblivious compare-and-set operator, and an oblivious compaction algorithm, the subORAM scheme described in §5 and formally defined in Figure 19 (see appendix) is secure according to Definition 2.*

Theorem 2. *Given a keyed cryptographic hash function, an oblivious compare-and-set operator, an oblivious sorting algorithm, an oblivious compaction algorithm, and an oblivious storage scheme (secure according to Definition 2), Snoopy, as described in §4 and formally defined in Figure 21 (see appendix), is secure according to Definition 1.*

All of the tools we use in the above theorems can be built from standard cryptographic assumptions. We prove both theorems in §B.

3 System overview

To motivate the design of our system, we begin by describing several solutions that do *not* work for our purposes.

Attempt #1: Scalable but not secure. Sharding is a straightforward way to achieve horizontal scaling. Each server maintains a separate ORAM for its data shard, and the client queries the appropriate server. This simple solution is insecure: repeated accesses to the same shard leaks query information. For example, if two clients query different servers, the attacker learns that they requested different objects.

Attempt #2: Secure but not scalable. To fix the above problem, we could remap an object to a different partition after it is accessed, similar to how single-server ORAMs remap objects after accesses [82, 93]. A central proxy running on a lightweight, trusted machine keeps a mapping of objects to

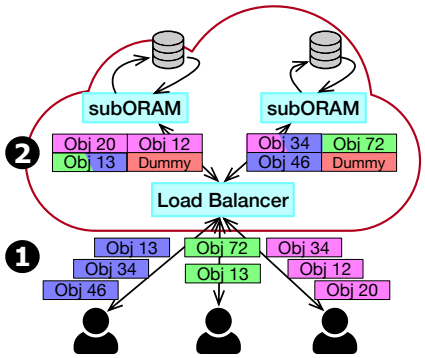


Figure 2. Secure distribution of requests in Snoopy. ❶ The load balancer receives requests from clients. ❷ At the end of the epoch, the load balancer generates a batch of requests for each subORAM, padding with dummy requests as necessary.

servers. The client sends its request to the proxy, which then accesses the server currently storing that object and remaps that object to a new server [8, 91]. While this solution is secure, this single proxy is a scalability bottleneck. Every request must use the most up-to-date mapping for security; otherwise, requests might fail and re-trying them will leak when the requested object was last accessed. Therefore, all requests must be serialized at the proxy, and so the proxy's throughput limits the system's throughput.

Our approach. We achieve the scalability of the first approach and the security of the second approach. To efficiently scale, we exploit characteristics of the high-throughput regime to develop new techniques that allow us to provide security *without* remapping objects across partitions. These techniques enable us to send equal-sized batches to each partition while both (1) hiding the mapping between requests and partitions (for security), and (2) ensuring that requests are distributed somewhat equally across partitions (for scalability).

3.1 System architecture

Snoopy's system architecture (Figure 2) consists of clients (running on private machines) and, in the public cloud, load balancers and subORAMs (running on hardware enclaves). All communication is encrypted using an authenticated encryption scheme with a nonce to prevent replay attacks. We establish all communication channels using remote attestation so that clients are confident that they are interacting with legitimate enclaves running Snoopy [5].

The role of the *load balancer* is to partition requests received during the last epoch into equally sized batches while providing security and efficiency (§4). In order to horizontally scale the load balancer, each load balancer must be able to operate independently and without coordination. The role of the *subORAM* is to manage a data partition, storing the current version of the data and executing batches of requests from the load balancers (§5). Snoopy can be deployed using any oblivious storage scheme for hardware enclaves [3, 65, 86] as a subORAM. However, our subORAM design is uniquely tailored to our target workload and end-to-end system design.

3.2 Real-world applications

Snoopy is valuable for applications that need a high-throughput object store for confidential data, including outsourced file storage [3], cloud electronic health records, and Signal's private contact discovery [60]. Privacy-preserving cryptocurrency light clients can also benefit from Snoopy. These allow lightweight clients to query full nodes for relevant transactions [62]. Maintaining many ORAM replicas is not enough to support high-throughput blockchains because each replica needs to keep up with the system state. As blockchains continue to increase in the throughput [84, 90], oblivious storage systems like Obladi [26] with a scalability bottleneck simply cannot keep up.

Snoopy can also enable private queries to a transparency log; for example, Alice could look up Bob's public key in a key transparency log [2, 63] without the server learning that she wants to talk to Bob. A key transparency log should support up to a billion users, making high throughput critical [35].

4 Oblivious load balancer

In this section, we detail the design of the load balancer, focusing on how batching can be used to hide the mapping between requests and subORAMs at low cost (§4.1), designing oblivious algorithms to efficiently generate batches while protecting the contents of the requests (§4.2), and scaling the load balancer across machines (§4.3).

4.1 Setting the batch size

To provide security, we need to ensure that constructing batches leaks no information about the requests. Specifically, we must guarantee that (1) the size of batches leaks no information, and (2) the process of constructing batches is similarly oblivious. We focus on (1) now and discuss (2) in §4.2. For security, we need to ensure that the batch size B depends only on public information visible to the attacker: namely, the number of requests R and number of subORAMs S, but not the contents of these requests. Therefore, we define B as a function $B = f(R,S)$ that outputs an efficient yet secure batch size for R requests and S subORAMs. Each subORAM will receive B requests. Because R is not fixed across epochs (requests can be bursty), B can also vary across epochs.

In choosing how to define this function f, we need to (1) ensure that requests are not dropped, and (2) minimize the overhead of dummy requests. Ensuring that requests are not dropped is critical for security: if a request is dropped, the client will retry the request, and an attacker who sees a client participate in two consecutive epochs may infer that a request was dropped, leaking information about request contents. Minimizing the overhead of dummy requests is important for scalability. A simple way to satisfy security

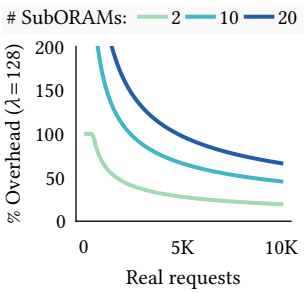


Figure 3. Dummy request overhead. A 50% overhead means for every two real requests there is one dummy request.

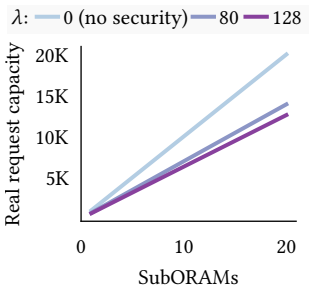


Figure 4. The total real request capacity of our system for an epoch, assuming ≤1K requests per subORAM per epoch.

would be to set $f(R,S) = R$; this ensures that even if all the requests are for the same object, no request was potentially dropped. However, this approach is not scalable because every subORAM would need to process a request for every client request. We refine this approach in two steps.

Deduplication to address skew. When assembling a batch of requests, the load balancer can ensure that all requests in a batch are for distinct objects by aggregating reads and writes for the same object (for writes, we use a "last write wins" policy) [26]. Deduplication allows us to combat workload skew. If the load balancer receives many requests for object A and a single request for object B, the load balancer only needs to send one request for object A and one request for object B. Deduplication simplifies the problem statement; we now need to distribute a batch of at most R *unique* requests across subORAMs. This reframing allows us to achieve security with high probability for $f(R,S) < R$ if we distribute objects randomly across subORAMs, as we now do not have to worry about the case where all requests are for the same object.

Choosing a batch size. Given R requests and S subORAMs, we need to find the batch size B such that the probability that any subORAM receives more than B requests is negligible in our security parameter λ. Like many systems that shard data, we use a hash function to distribute objects across subORAMs, allowing us to recast the problem of choosing B as a balls-into-bins problem [76]: we have R balls (requests) that we randomly toss into S bins (subORAMs), and we must find a bin size B (batch size) such that the probability that a bin overflows is negligible. We add balls (dummy requests) to each of the S bins such that each bin contains exactly B balls.

Using the balls-into-bins model, we can start to understand how we expect R and S to affect B. As we add more balls to the system ($R\uparrow$), it becomes more likely for the balls to be distributed evenly over every bin, and the ratio of dummy balls to original balls decreases. Conversely, as we add more bins to the system ($S\uparrow$), we need to proportionally add more dummy balls. We validate this intuition in Figure 3 and Figure 4. Figure 3 shows that as the total number of requests R increases, the percent overhead due to dummy requests decreases. Thus larger batch sizes are preferable, as they minimize the overhead introduced by dummy requests. Figure 4 illustrates how adding more subORAMs increases the total request capacity of Snoopy, but at a slower rate than a plaintext system. Adding subORAMs helps Snoopy scale by breaking data into partitions, but adding subORAMs is not free, as it increases the dummy overhead.

We prove that the following f for setting batch size B guarantees negligible overflow probability in §A:

Theorem 3. *For any set of R requests that are distinct and randomly distributed, number of subORAMs S, and security parameter λ, let $\mu = R/S$, $\gamma = -\log(1/(S \cdot 2^{\lambda}))$, and $W_0(\cdot)$ be branch 0 of the Lambert W function [23]. Then for the following function $f(R,S)$ that outputs a batch size, the probability that a request is dropped is negligible in λ:*

$$f(R,S) = \min\left(R,\ \mu \cdot \exp\left[W_0\left(e^{-1}(\gamma/\mu - 1)\right) + 1\right]\right).$$

Proof intuition. For a single subORAM s, let $X_1,\ldots,X_R \in \{0,1\}$ be independent random variables where X_i represents request i mapping to s. Then, $\Pr[X_i = 1] = 1/S$. Next, let the random variable $X = \sum_{i=1}^{R} X_i$ represent the total number of requests that hashed to s. We use a Chernoff bound to upper-bound the probability that there are more than k requests to a single subORAM, $\Pr[X \geq k]$. In order to upper-bound the probability of overflow for *all* subORAMs, we use the union bound and solve for the smallest k that results in an upper bound on the probability of overflow negligible in λ. In order to solve for k, we coerce the inequality into a form that can be solved with the Lambert W function, which is the inverse relation of $f(w) = we^{w}$, i.e., $W(we^{w}) = w$ [23]. When $f(R,S) = R$, the overflow probability is zero, and so we can safely upper-bound $f(R,S)$ by R. We target the high-throughput case where R is large, in which case our bound is less than R.

We now explain how Theorem 3 applies to Snoopy. For security, it is important that an attacker cannot (except with negligible probability) choose a set of requests that causes a batch to overflow. Thus Snoopy needs to ensure that requests chosen by the attacker are transformed to a set of requests that are distinct and randomly distributed across subORAMs. Snoopy ensures that requests are distinct through deduplication and that requests are randomly distributed by using a keyed hash function where the attacker does not know the key. Because the keyed hash function remains the same across epochs, Snoopy must prevent the attacker from learning which request is assigned to which subORAM during execution (otherwise, the attacker could use this information to construct requests that will overflow a batch). Snoopy does this by ensuring that each subORAM receives the same number of requests and by obliviously assigning requests to the correct subORAM batch (§4.2.2). Theorem 3 allows us to choose a batch size that is less than R in the high-throughput setting (for scalability) while ensuring that the probability

that an attacker can construct a batch that causes overflow is cryptographically negligible. Thus Snoopy achieves security for *all workloads*, including skewed ones.

The bound we derive is valuable in applications beyond Snoopy where there are a large number of balls and it is important that the overflow probability is very small for different numbers of balls and bins. Our bound is particularly useful in the case where the overflow probability must be negligible in the security parameter as opposed to an application parameter (e.g. the number of bins) [7, 66, 76, 77].

4.2 Oblivious batch coordination

As with other components of the system, the load balancer runs inside a hardware enclave, and so we must ensure that its memory accesses remain independent of request content. The load balancer runs two algorithms that must be oblivious: generating batches of requests (§4.2.2), and matching subORAM responses to client requests (§4.2.3).

Practically, designing oblivious algorithms requires ensuring that the memory addresses accessed do not depend on the data; often this means that the access pattern is fixed and depends only on public information (alternatively, access patterns might be randomized). The data contents remain encrypted and inaccessible to the attacker, and only the pattern in which memory is accessed is visible. We build our algorithms on top of an oblivious "compare-and-set" operator that allows us to copy a value if a condition is true without leaking if the copy happened or not.

4.2.1 Background: oblivious building blocks. We first provide the necessary background for two oblivious building blocks from existing work that we will use in our algorithms.

Oblivious sorting. An oblivious sort orders an array of n objects without leaking information about the relative ordering of objects. We use bitonic sort, which runs in time $O(n\log^2 n)$ and is highly parallelizable [6]. Bitonic sort accesses the objects and performs compare-and-swaps in a *fixed, predefined order*. Since its access pattern is independent of the final order of the objects, bitonic sort is oblivious.

Oblivious compaction. Given an array of n objects, each of which is tagged with a bit $b \in \{0,1\}$, oblivious compaction removes all objects with bit $b=0$ without leaking information about which objects were kept or removed (except for the the total number of objects kept). We use Goodrich's algorithm, which runs in time $O(n\log n)$ and is *order-preserving*, meaning that the relative order of objects is preserved after compaction [34]. Goodrich's algorithm accesses array locations in a fixed order using a $\log n$-deep routing network that shifts each element a fixed number of steps in every layer.

4.2.2 Generating batches of requests. Generating fixed-size batches *obliviously* requires care. It is not enough to simply pad batches with a variable number of dummy requests, as this can leak the number of real requests in each batch. Instead, we must pad each batch with the right number of dummy requests *without revealing the exact number of dummy requests added to each batch.* To solve this problem, we obliviously generate batches in three steps, which we show in Figure 5: ❶ we first assign client requests to subORAMs according to their requested object; ❷ we add the maximum number of dummy requests to each subORAM; ❸ we construct batches with those extra dummies; and ❹ we filter out unnecessary dummies.

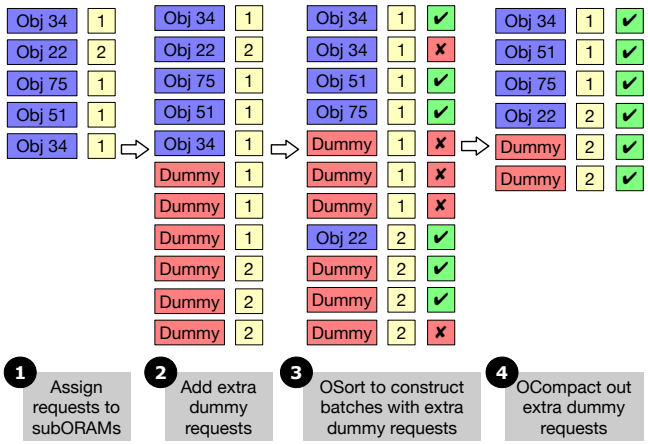


Figure 5. Generating batches of requests at the load balancer.

First (❶), we scan through the list of client requests. For each client request, we compute the subORAM ID by hashing the object ID, and we store it with the client request. Second (❷), we append the maximum number of dummy requests for each subORAM, $B = f(R,S)$ to the end of the list. These dummy requests all have a tag bit $b=1$. Third (❸), we group real and dummy requests into batches by subORAM. We do this by obliviously sorting the lists of requests, setting the comparison function to order first by subORAM (to group requests into subORAM batches), then by tag bit b (to push the dummies to the end of the batches), and then by object ID (to place duplicates next to each other). Finally (❹), to choose which requests to keep and which to remove, we iterate through the sorted request list again. We keep a counter x of the number of distinct requests seen so far for the current subORAM. We securely update the counter by performing an oblivious compare-and-set for each request, ensuring that access patterns don't reveal when the counter is updated. If $x < B$ and the request is not a duplicate (i.e. it is not preceded by a request for the same object), we set bit $b=1$ (otherwise $b=0$). To filter out unnecessary dummy requests and duplicates, we obliviously compact by bit b, leaving us with a B-sized batch for each subORAM.

The algorithm is oblivious because it only relies on linear scans and appends (both are data-independent) and our oblivious building blocks. The runtime is dominated by the cost of oblivious sorting and compaction.

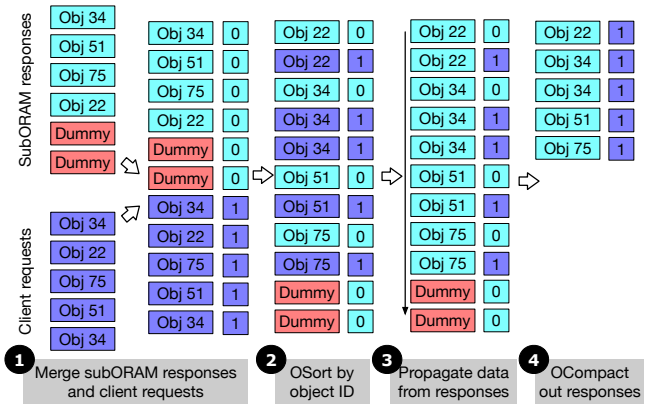


Figure 6. Mapping subORAM responses to client requests at the load balancer.

4.2.3 Mapping responses to client requests.

Once we receive the batches of responses from the subORAMs, we need to send replies to clients. This requires mapping the data from subORAM responses to the original requests, making sure that we propagate data correctly to duplicate responses and that we ignore responses to dummy requests. We accomplish this obliviously in four steps, which we show in Figure 6: ❶ we merge together the client requests and the subORAM responses and then sort the list; ❷ we sort the merged list to group requests with responses; ❸ we propagate data from the responses to the original requests; and ❹ we filter out the now unnecessary subORAM responses.

The load balancer takes as input two lists: a list of subORAM responses and a list of client requests. First (❶), we merge the two lists, tagging the subORAM responses with a bit $b=0$ and the client requests with $b=1$. Second (❷), we sort this combined list by object ID and then, to break ties, by the tag bit b. Breaking ties by the tag bit b arranges the data so that we can easily propagate data from subORAM responses to requests. Third (❸), we iterate through the list, propagating data in objects with the tag bit $b=0$ (the subORAM responses) to the following object(s) with the tag bit $b=1$ (the client requests). As we iterate through the list, we keep track of the last object we have seen with $b=1$, prev (i.e. the last subORAM response we've scanned over). Then, for the current object curr, we copy the contents of prev into the curr if $b=0$ for curr (it's a request). Any requests following a response must be for the same object because every request has a corresponding response and we sort by object ID. Note that dummy responses will not have a corresponding client request. Finally (❹), we need to filter down the list to include only the client requests. We do this using oblivious compaction, removing objects with the tag bit $b=0$ (the subORAM responses). Note that, in order to respond to a request, we need to map a client request to the original network connection; we can do this by keeping a pointer to the connection with the request data.

This procedure is oblivious because it relies only on oblivious building objects as well as concatenating two lists and a linear scan, both of which are data-independent. As in the algorithm for generating batches, the runtime is dominated by the cost of oblivious sorting and oblivious compaction.

4.3 Scaling the load balancer

Our load balancer design scales horizontally; it is both correct and secure to add load balancers without introducing additional coordination costs. Clients randomly choose one load balancer to contact, and then each load balancer batches requests independently. This is a significant departure from prior work where a centralized proxy receives all client requests and must maintain dynamic state relevant to all requests [8, 26, 85, 91]. SubORAMs execute load balancer batches in a fixed order, and within a single load balancer, we aggregate reads and writes using a "last-write-wins" policy.

Adding load balancers eliminates a potential bottleneck, but is not entirely free. Because (1) load balancers do not coordinate to deduplicate requests and (2) subORAMs assume that a batch contains distinct requests, subORAMs cannot combine batches from different load balancers. Our subORAM must scan over all stored objects to process a single batch (§5). As a result, if there are L load balancers, each subORAM must perform L scans over the data every epoch.

5 Throughput-optimized subORAM

Many ORAMs target asymptotic complexity, often at the expense of concrete cost. In contrast, recent work has explored how to leverage *linear scans* to build systems that can achieve better performance for expected workloads than their asymptotically more efficient counterparts [27, 29]. We take a similar approach to design a high-throughput subORAM optimized for hardware enclaves. We exploit the fact that, due to Snoopy's design, each subORAM stores a relatively small data partition and receives a batch of distinct requests. In this setting, using a *single linear scan* over the data partition to process a batch is concretely efficient in terms of amortized per-request cost.

We draw inspiration from Signal's private contact discovery protocol [60]. There, the client sends its contacts to an enclave, and the enclave must determine which contacts are Signal users without leaking the client's contacts. Their solution employs an *oblivious hash table*. The core idea is that the enclave performs some expensive computation to construct a hash table such that the construction access patterns don't leak the mapping of contacts to buckets. Once this hash table is constructed, the enclave can directly access the hash bucket for a contact without the memory access pattern revealing which contact was looked up. Note that obliviousness only holds if (1) the enclave performs a lookup for each contact at most once, and (2) the enclave scans the entire bucket (to avoid revealing the location of the contact accessed inside the bucket). With this tool, private contact discovery is

straightforward: the enclave constructs an oblivious hash table for the client's contacts and then scans over every Signal user, looking up each Signal user in the contact hash table.

Signal's setting is similar to ours: instead of a set of contacts, we have a batch of distinct requests, and instead of needing to find matches with the Signal users, we need to find the stored objects corresponding to requests. However, Signal's approach has some serious shortcomings when applied to our setting. First, their hash table construction takes $O(n^2)$ time for n contacts. While this complexity is acceptable when n is the size of a user's contacts list (relatively small), it is prohibitively expensive for batches with thousands of requests. Second, they do not size their buckets to prevent overflow. Overflows can leak information about bucket contents, and attempting to recover causes further leakage [16, 54].

Choosing an oblivious hash table. We need to identify an oblivious hash table that is efficient and secure in our setting. A natural first attempt to solve the overflow problem is to use the number of requests that hash to each bucket to set the bucket size dynamically. This simple solution is insecure: the attacker can infer the probability that an object was requested based on the size of the bucket that object hashes to.

Instead, we need to set the bucket size so that the overflow probability is cryptographically negligible. This provides the security property we want, and is exactly the problem that we solved in the load balancer, where we separated requests into "bins" such that the probability that any "bin" overflows is negligible. Using our load balancer approach also reduces construction cost from $O(n^2)$ to $O(n \operatorname{polylog} n)$. However, while this solution works well at the load balancer, it becomes expensive when applied to the subORAM. Recall that to perform an oblivious lookup, we must scan the entire bucket that might contain a request, and so we want buckets to be as small as possible. Unfortunately, decreasing the bucket size results in substantial dummy overhead. This overhead was the reason for making our batches as large as possible at the load balancer (Figure 3). In our subORAM, we want to keep the dummy overhead low *and* have a small bucket size.

To achieve both these properties, we identify *oblivious two-tier hash tables* as a particularly well-suited to our setting [16]. Chan et al. show how to size buckets such that overflow requests are placed into a second hash table, allowing us to have both low dummy overhead and a small bucket size: for batches of 4,096 requests, buckets in a two-tier hash table are ~10× smaller than their single-tier counterparts. Construction now requires two oblivious sorts, one for each tier, but is still much faster than Signal's approach, both asymptotically and concretely for our expected batch sizes. We refer the reader to Chan et al. for the details of oblivious construction, oblivious lookups, and the security analysis [16].

Processing a batch of requests. We now describe how to leverage an oblivious two-tier hash table to obliviously process a batch of requests (Figure 7). First (❶), when the batch

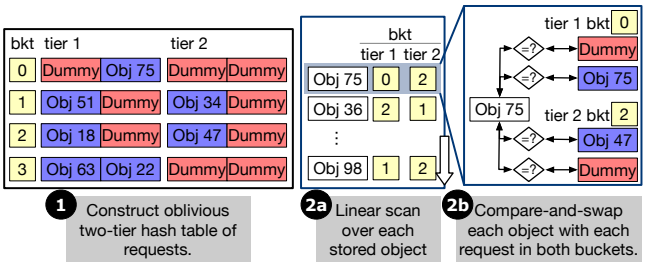


Figure 7. Processing a batch of requests at a subORAM.

of requests arrives, we construct the oblivious two-tier hash table as described above. To avoid leaking the relationship between requests across batches, for every batch we sample a new key (unknown to the attacker) for the keyed hash function assigning objects to buckets. Second (❷), we iterate through the stored objects. For each object obj, we perform an oblivious hash table lookup. A lookup requires hashing obj.id in order to find the corresponding bucket in both hash tables and then scanning the entire bucket; this scan is necessary to hide the specific object being looked up. For every request req scanned, we perform an oblivious compare-and-set to update either the req in the hash table or the obj in subORAM storage depending on (1) whether req.id matches obj.id, and (2) whether req is a read or write. By conditioning the oblivious compare-and-set on the request type and performing it twice (once on the contents of req and once on the contents of obj), we hide whether the request is a read or a write.

Finally, we scan through every hash table bucket, marking real requests with tag bit $b = 1$ and dummies with $b = 0$. We then use oblivious compaction to filter out the dummies, leaving us with real entries to send back to the load balancer.

6 Planner

Our Snoopy planner takes as input a data size D, minimum throughput X_{Sys}, maximum latency L_{Sys}, and outputs a configuration (number of load balancers and subORAMs) that minimizes system cost. As the search space is large, we rely on heuristics and make simplifying assumptions to approximate the optimal configuration. We derive three equations capturing the relationship between our core system parameters: the epoch length T, number of objects N, number of subORAMs S, and number of load balancers B.

To estimate throughput for some epoch time T, we observe that, on average, we must be able to process all requests received during the epoch in time $\leq T$ (otherwise, the set of outstanding requests continues growing). We can pipeline the subORAM and load balancer processing such that the upper bound on the requests we can process per epoch is determined by either the load balancer or subORAM processing time, depending on which is slower. Adding load balancers decreases the work done at each load balancer, but each subORAM must process a batch of requests from every load balancer. Let $L_{\text{LB}}(R,S)$ be the time it takes a load

balancer to process R requests in a system with S subORAMs, and let $L_S(R,S,N)$ be the time it takes a subORAM to process a batch of R requests with N stored objects. We then derive:

$$T \geq \max[L_{\text{LB}}(X_{\text{Sys}} \cdot T/B,S),\ B \cdot L_S(f(X_{\text{Sys}} \cdot T/B,S),N)] \quad (1)$$

Requests will arrive at different times and have to wait until the end of the current epoch to be serviced, and so on average, if the timing of requests is uniformly distributed, requests will wait on average $T/2$ time to be serviced. The time to process a batch is upper-bounded by T at both the subORAM and the load balancer, and so:

$$L_{\text{Sys}} \leq 5T/2 \quad (2)$$

Let C_{LB} be the cost of a load balancer and C_S be the cost of a subORAM. We then compute the system cost C_{Sys}:

$$C_{\text{Sys}}(B,S) = B \cdot C_{LB} + S \cdot C_S \quad (3)$$

Our planner uses these equations and experimental data to approximate the cheapest configuration meeting performance requirements. While our planner is useful for selecting a configuration, it does not provide strong performance guarantees, as our model makes simplifying assumptions and ignores subtleties that could affect performance (e.g. our simple model assumes that requests are uniformly distributed). Our planner is meant to be a starting point for finding a configuration. Our design could be extended to provide different functionality; for example, given a throughput, data size, and cost, output a configuration minimizing latency.

7 Implementation

We implemented Snoopy in ~7,000 lines of C++ using the OpenEnclave framework v0.13 [70] and Intel SGX v2.13. We use gRPC v1.35 for communication and OpenSSL for cryptographic operations. Our bitonic sort [6] and oblivious compaction [34] implementations set the size of oblivious memory to the register size. We use Intel's AVX-512 SIMD instructions for oblivious compare-and-swaps and compare-and-sets. Our implementation is open-source [1].

Reducing enclave paging overhead. The size of the protected enclave memory (EPC) is limited and enclave memory pages that do not fit must be paged in when accessed, which imposes high overheads [71]. The data at a subORAM often does not fit inside the EPC, so to reduce the latency to page in from untrusted memory, we rely on a shared buffer between the enclave and the host. A host loader thread fills the buffer with the next objects that the linear scan will read. This eliminates the need to exit and re-enter the enclave to fetch data, dramatically reducing linear scan time. The enclave encrypts objects (for confidentiality) and stores digests of the contents inside the enclave (for integrity). This approach has been explored in prior enclave systems [74, 75].

	Redis [81]	Obladi [26]	Oblix [65]	**Snoopy**
Oblivious	✗	✓	✓	✓
No trusted proxy	✓	✗	✓	✓
High throughput	✓	✓	✗	✓
Throughput scales with machines	✓	✗	✗	✓

Table 8. Comparison of baselines based on security guarantees (oblivious), setup (no trusted proxy), and performance properties (high throughput and throughput scales).

8 Evaluation

To quantify how Snoopy overcomes the scalability bottleneck in oblivious storage, we ask:

1. How does Snoopy's throughput scale with more compute, and how does it compare to existing systems? (§8.2)
2. How does adding compute resources help Snoopy reduce latency and scale to larger data sizes? (§8.3)
3. How do Snoopy's individual components perform? (§8.4)
4. Given performance and monetary constraints, what is the optimal way to allocate resources in Snoopy? (§8.5)

Experiment Setup. We run Snoopy on Microsoft Azure, which provides support for Intel SGX hardware enclaves in the DCsv2 series. For the load balancers and subORAMs, we use `DC4s_v2` instances with 4-core Intel Xeon E-2288G processors with Intel SGX support and 16GB of memory. For clients, we use `D16d_v4` instances with 16-core Intel Xeon Platinum 8272CL processors and 64GB of memory. We choose these instances for their comparatively high network bandwidth. We evaluate our baselines Redis [81] on `D4d_v4` instances, Obladi [26] on `D32d_v4` for the proxy and `D16d_v4` for the storage server, and Oblix on the same `DC4s_v2` instances as our subORAMs. For benchmarking, we use a uniform request distribution. This choice is only relevant for our Redis baseline; the oblivious security guarantees of Snoopy and other oblivious storage systems ensure that the request distribution does not impact their performance. Unless otherwise specified, we set the object size to 160 bytes (same as Oblix [65]).

8.1 Baselines

We compare Snoopy to three state-of-the-art baselines: Obladi [26] is a batched, high-throughput oblivious storage system, Oblix [65] efficiently leverages enclaves for oblivious storage, and Redis [81] is a widely used plaintext key-value store. Each baseline provides a different set of security guarantees and performance properties (Table 8).

Obladi. Obladi [26] uses batching and parallelizes RingORAM [82] to achieve high throughput. While Obladi also uses batching to improve throughput, its security model is different, as it uses a single trusted proxy rather than a

hardware enclave. The trusted proxy model has two primary drawbacks: (1) the trusted proxy cannot be deployed in the untrusted cloud (desirable for convenience and scalability), and (2) the proxy is a central point of attack in the system (an attacker that compromises the proxy learns the queries of every user in the system). Practically, using a trusted proxy rather than a hardware enclave means the proxy does not have to use oblivious algorithms. Designing an oblivious algorithm for Obladi's proxy is not straightforward and would likely introduce significant overhead. Further, Obladi's trusted proxy is a compute bottleneck that cannot be horizontally scaled securely without new techniques, and so we only measure Obladi with two machines (proxy and storage server). We configure Obladi with a batch size of 500.

Oblix. Oblix [65] uses hardware enclaves and provides security guarantees comparable to ours. However, Oblix optimizes for latency rather than throughput; requests are sequential, and, unlike Obladi, Oblix does not employ batching or parallelism. Like Obladi, Oblix cannot securely scale across machines. We measure performance using Oblix's DORAM implementation and simulate the overhead of recursively storing the position map (as in §VI.A of [65]).

Redis. To measure the overhead of security (obliviousness), we compare Snoopy to an insecure baseline Redis [81], a popular unencrypted key-value store. In Redis, the server can directly see access patterns and data contents. We benchmark a Redis cluster using its own `memtier` benchmark tool [64], enabling client pipelining to trade latency for throughput. We expect it to achieve a much higher throughput than Snoopy.

8.2 Throughput scaling

Figure 9a shows that adding more machines to Snoopy improves throughput. We measure throughput where the average latency is less than 300ms, 500ms, and 1s. We start with 4 machines (3 subORAMs and 1 load balancer) and scale to 18 machines (13 subORAMs and 5 load balancers for 1s latency; 15 subORAMs and 3 load balancers for 500ms/300ms latency). For 2M objects, Snoopy uses 18 machines to process 68K reqs/sec with 300ms latency, 92K reqs/sec with 500ms latency, and 130K reqs/sec with 1s latency. Each additional machine improves throughput by 8.6K reqs/sec on average for 1s latency. Relaxing the latency requirement improves throughput because we can group requests into larger batches, reducing the overhead of dummy requests.

We generate Figure 9a by measuring throughput with different system configurations and plotting the highest throughput configuration for each number of machines. We start with 4 machines rather than 2 because we need to partition the 2M objects to meet our 300ms latency requirement due to the subORAM linear scan (recall Equation (2) would require a subORAM to process a batch in ≤ 120ms). Both the load balancer and subORAM are memory-bound, as the EPC size is limited and enclave paging costs are high (§7).

Snoopy achieves higher throughput than Oblix (1,153 reqs/sec) and Obladi (6,716 reqs/sec) as we increase the number of machines. For 300ms, Snoopy outperforms Oblix with $\geq$5 machines and Obladi with $\geq$6 machines, and for 500ms and 1s, Snoopy outperforms Oblix and Obladi for all configurations. Oblix and Obladi beat Snoopy with a small number of machines for low latency requirements because our subORAM performs a linear scan over subORAM data whereas Oblix and Obladi only incur polylogarithmic access costs, allowing them to handle larger data sizes on a single machine. Snoopy can scale to larger data sizes by adding more machines (§8.3).

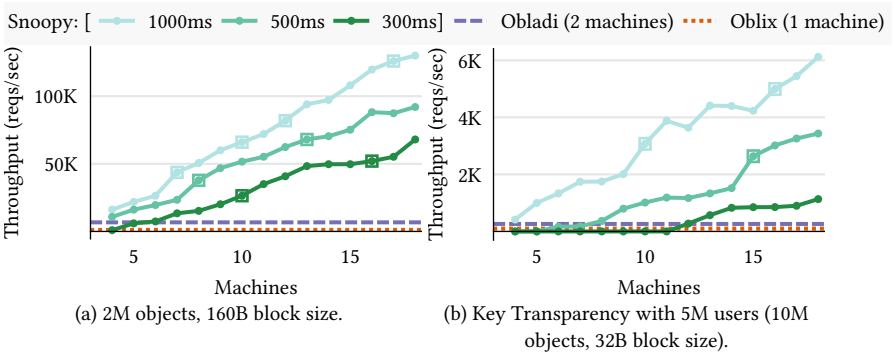
Comparison to Redis. To show the overhead of obliviousness, we also measure the throughput of Redis for 2M 160-byte objects with an increasing cluster size. For 15 machines, Redis achieves a throughput of 4.2M reqs/sec, 39.1× higher than Snoopy when configured with 1s latency. Because we pipeline Redis aggressively in order to maximize throughput, the mean Redis latency is <800ms.

Application: key transparency. Figure 9b shows throughput for parameter settings that support key transparency (KT) [2, 63] for 5 million users. Due to the security guarantees of oblivious storage, an application's performance does not depend on its workload (i.e. request distribution), but only on the parameter settings. In KT, to look up Bob's key, Alice must retrieve (1) Bob's key, (2) the signed root of the transparency log, and (3) a proof that Bob's key is included in the transparency log (relative to the signed root) [63]. This inclusion proof is simply a Merkle proof. Thus, for n users, Alice must make $\log_2 n+1$ ORAM accesses (Alice can request the signed root directly). Figure 9b shows that by adding machines, Snoopy scales to support high throughput for KT. At 18 machines (15 subORAMs and 3 load balancers), Snoopy can process 1.1K reqs/sec with 300ms latency, 3.2K reqs/sec with 500ms latency, and 6.1K reqs/sec with 1s latency. Note that the throughput in Figure 9b is much lower than Figure 9a because each KT operation requires 24 ORAM accesses.

Oblix as a subORAM. In Figure 10, we run Oblix [65] as a subORAM instead of Snoopy's throughput-optimized subORAM (§5). Snoopy's load balancer design enables us to securely scale Oblix beyond a single machine, achieving 15.6× higher throughput with Snoopy-Oblix for 17 machines with a max latency of 500ms (18K reqs/sec) than vanilla, single-machine Oblix (1.1K reqs/sec). The spike in throughput between 8 and 9 machines is due to sharding the data such that two instead of three layers of recursive lookups are required for every ORAM access. Snoopy-Oblix's performance also illustrates the value of our subORAM design; using our throughput-optimized subORAM (Figure 9a) improves throughput by 4.85× with 17 machines and 500ms latency.

8.3 Scaling for latency and data size

While Snoopy is designed specifically for throughput scaling (§8.2), adding machines to Snoopy can have other benefits if



(a) 2M objects, 160B block size.

(b) Key Transparency with 5M users (10M objects, 32B block size).

Figure 9. Snoopy achieves higher throughput with more machines. Boxed points denote when a load balancer is added instead of a subORAM. Oblix and Obladi cannot securely scale past 1 and 2 machines, respectively.

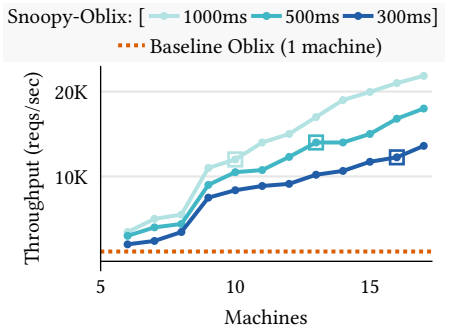
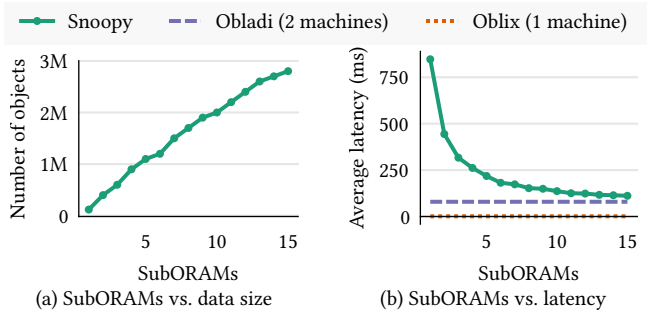


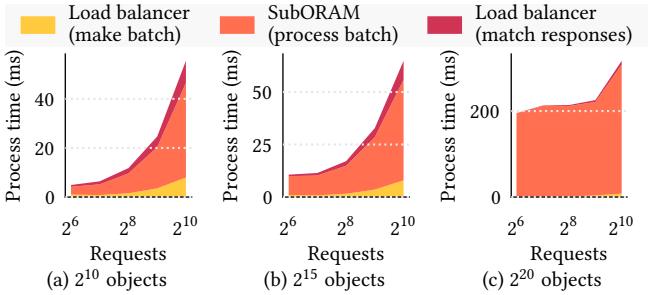
Figure 10. Throughput of Snoopy using Oblix [65] as a subORAM (2M objects, 160B block size). We measure throughput with different maximum average latencies.



(a) SubORAMs vs. data size

(b) SubORAMs vs. latency

Figure 11. (a) Adding more subORAMs allows for increasing the data size while keeping the average response time under 160ms (RTT from US to Europe). (b) Adding more subORAMs reduces latency. Snoopy is running 1 load balancer and storing 2M objects.



(a) 2^{10} objects

(b) 2^{15} objects

(c) 2^{20} objects

Figure 12. Breakdown of time to process one batch for different data sizes (one load balancer and one subORAM).

the load remains constant. We show how scaling can be used to both reduce latency and tolerate larger data sizes under constant load in Figure 11. Figure 11a illustrates how adding more subORAMs enables us to increase the number of objects Snoopy can store while keeping average response time under 160ms (the round-trip time from the US to Europe). The number of subORAMs required scales linearly with the data size because of the linear scan every epoch. Adding a subORAM allows us to store on average 191K more objects, and with 15 subORAMs, we can store 2.8M objects.

Figure 11b shows how adding subORAMs reduces latency when data size and load are fixed: for 2M objects, the mean latency is 847ms with 1 subORAM and 112ms with 15 subORAMs. Adding subORAMs parallelizes the linear scan across more machines, but has diminishing returns on latency because the dummy request overhead also increases when we add subORAMs (Figure 3). As expected, Oblix achieves a substantially lower latency (1.1ms) because it uses a tree-based ORAM and processes requests sequentially. Obladi achieves a latency of 79ms with batch size 500.

8.4 Microbenchmarks

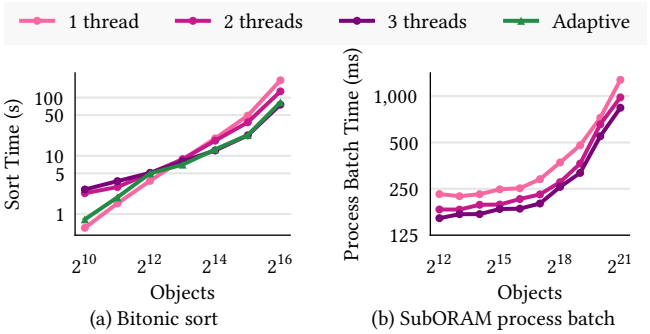
Breakdown of batch processing time. Figure 12 illustrates how time is spent processing a batch of requests as batch size increases. As batch size increases, the load balancer time also increases, as the load balancer must obliviously generate batches. The subORAM time is largely dependent on the data size, as the processing time is dominated by the linear scan over the data. The subORAM batch processing time jumps between 2^{15} and 2^{20} objects due to the cost of enclave paging.

Sorting parallelism. In Figure 13a, we show how parallelizing bitonic sort across threads reduces latency, especially for larger data sizes. For smaller data sizes, the coordination overhead actually makes it cheaper to use a single thread, and so we adaptively switch between a single-threaded and multi-threaded sort depending on data size. Parallelizing bitonic sort improves load balancer and subORAM performance.

SubORAM Parallelism. Similarly, in Figure 13b, we show how additional cores can be used to reduce subORAM batch processing time. We rely on a host thread to buffer in the encrypted data in the linear scan over the all objects in the subORAM (§7), and we can use the remaining cores to parallelize both the hash table construction and linear scan.

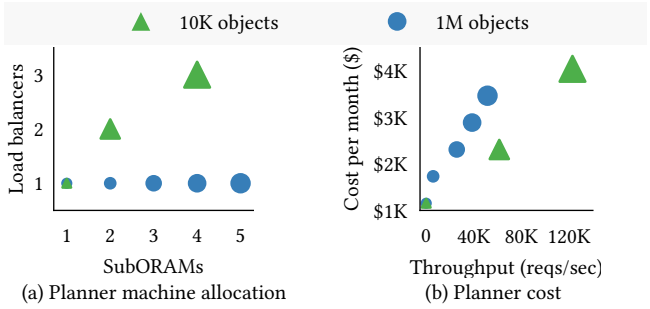
8.5 Planner

In Figure 14, we use our planner to find the optimal resource allocation for different performance requirements. Figure 14a



(a) Bitonic sort (b) SubORAM process batch

Figure 13. (a) Parallelizing bitonic sort across multiple threads. (b) Parallelizing batch processing at the subORAM across multiple enclave threads (batch size 4K requests).



(a) Planner machine allocation (b) Planner cost

Figure 14. Optimal system configuration as throughput requirements increase for different data sizes (max latency 1s). Larger dot sizes represent higher throughput requirements. We show a subset of configurations from our planner in order to illustrate the overall trend of how adding machines best improves throughput.

shows the optimal number of subORAMs and load balancers to handle an increasing request load for different data sizes with 1s average latency. To support higher throughput levels, deployments with larger data sizes benefit from a higher ratio of subORAMs to load balancers, as partitioning across subORAMs parallelizes the linear scan over stored objects. In Figure 14b, we show how increasing throughput requirements affects system cost for different data sizes. Increasing data increases system cost: for ~$4K/month, we can support 51.6K reqs/sec for 1M objects and 122.9K reqs/sec for 10K objects. To compute these configurations, the planner takes as input microbenchmarks for different batch sizes and data sizes. Because we cannot benchmark every possible batch and data size, we use the microbenchmarks for the closest parameter settings. Our planner's estimates could be sharpened further by running microbenchmarks at a finer granularity.

9 Discussion

Fault tolerance and rollback protection. Data loss in Snoopy can arise through node crashes and malicious rollback attacks. Many modern enclaves are susceptible to rollback attacks where, after shutdown, the attacker replaces the latest sealed data with an older version without the enclave detecting this change [73]. Prior work has explored how to defend against such attacks [11, 61]. Fault tolerance and rollback prevention are not the focus of this paper, and so we only briefly describe how Snoopy could be extended to defend against data loss. All techniques are standard. Load balancers are stateless; we thus exclusively consider subORAMs. We propose to use a quorum replication scheme to replicate data to $f+r+1$ nodes where f is the maximum number of nodes that can fail by crashing and r the maximum number of nodes that can be maliciously rolled back. Systems like ROTE [61] or SGX's monotonic counter provide a trusted counter abstraction that can be used to detect which of the received replies corresponds to the most recent epoch. The performance overhead of rollback protection would depend on the trusted counter mechanism employed, but Snoopy only invokes the trusted counter once per epoch.

Next-generation SGX enclaves. While current SGX enclaves can only support a maximum EPC size of 256MB, upcoming third-generation SGX enclaves can support EPC sizes up to 1TB [46]. This new enclave would not affect Snoopy's core design, but could improve performance by reducing the time for the per-epoch linear scan in the subORAM. With improved subORAM performance, Snoopy might need fewer subORAMs for the same amount of data, affecting the configurations produced by the planner (§8.5).

Private Information Retrieval (PIR). Snoopy's techniques can also be applied to the problem of private information retrieval (PIR) [21, 22]. A PIR protocol allows a client to retrieve an object from a storage server without the server learning the object retrieved. One fundamental limitation of PIR is that, if the object store is stored in its original form, the server must scan the entire object store for each request.

Snoopy's techniques can help overcome this limitation. We can replace the subORAMs with PIR servers, each of which stores a shard of the data. Our load balancer design then makes it possible to obliviously route requests to the PIR server holding the correct shard of the data. "Batch" PIR schemes that allow a client to fetch many objects at roughly the server-side cost of fetching a single object are well-suited tor our setting, as the load balancer is already aggregating batches of requests [42, 47]. Existing systems develop relevant batching [4, 40] and preprocessing [51] techniques.

10 Related work

We summarize relevant existing work, focusing on (1) oblivious algorithms designed for hardware enclaves, (2) ORAM parallelism, (3) distributing an ORAM across machines, and (4) balls-into-bins bounds for maximum load.

ORAMs with secure hardware. Existing research on oblivious computation using hardware enclave primarily targets latency. Oblix [65], ZeroTrace [86], Obliviate [3], Pyramid

ORAM [24], and POSUP [45] do not support concurrency. Snoopy, in contrast, optimizes for throughput and leverages batching for security and scalability. ObliDB [30] supports SQL queries by integrating PathORAM with hardware enclaves, but uses an oblivious memory pool unavailable in Intel SGX. GhostRider [58] and Tiny ORAM [32] use FPGA prototypes designed specifically for ORAM. While no general-purpose, enclave-based ORAM supports request parallelism, MOSE [44] and Shroud [59] leverage data parallelism to improve the latency of a single request on large datasets. MOSE runs CircuitORAM [17] inside a hardware enclave and distributes the work for a single request across multiple cores. Shroud instead parallelizes Binary Tree ORAM across many secure co-processors by accessing different layers of the ORAM tree in parallel. Shroud uses data parallelism to optimize for latency and data size; throughput scaling is still limited because requests are processed sequentially.

Supporting ORAM parallelism. A rich line of work explores executing multiple client requests in parallel at a single ORAM server. Each requires some centralized component(s) that eventually bottlenecks scalability. PrivateFS [101] and ConcurORAM [14] coordinate concurrent requests to shared data using an encrypted query log on top of a hierarchical ORAM or a tree-based ORAM, respectively. This query log quickly becomes a serialization bottleneck. TaoStore [85] and Obladi [26] similarly rely on a trusted proxy to coordinate accesses to PathORAM and RingORAM, respectively. Taostore processes requests immediately, maintaining a local subtree to securely handle requests with overlapping paths. Obladi instead processes requests in batches, amortizing the cost of reading/writing blocks over multiple requests. Batching also removes any potential timing side-channels; while TaoStore has to time client responses carefully, Obladi can respond to all client requests at once, just as in Snoopy.

PRO-ORAM [95], a read-only ORAM running inside an enclave, parallelizes the shuffling of batches of $\sqrt{N}$ requests across cores, offering competitive performance for read workloads. Snoopy, in contrast, supports both reads and writes.

A separate, more theoretical line of work considers the problem of Oblivious Parallel RAMs (OPRAMs), designed to capture parallelism in modern CPUs. Initiated by Boyle et al. [10], OPRAMs have been explored in subsequent work [15–18] and expanded to other models of parallelism [79].

Scaling out ORAMs. Several ORAMs support distributing compute and/or storage across multiple servers. Oblivistore [91] distributes partitions of SSS-ORAM [92] across machines and leverages a load balancer to coordinate accesses to these partitions. This load balancer, however, does not scale and becomes a central point of serialization. CURIOUS [8] is similar, but uses a simpler design that supports different subORAMs (e.g. PathORAM). CURIOUS distributes storage but not compute; a single proxy maintains the mapping of blocks between subORAMs and runs the subORAM clients, which bottlenecks scalability. In contrast, Snoopy distributes both compute and storage and can scale in the number of subORAMs *and* load-balancers. Moreover, Snoopy remains secure when an attacker can see client response timing, unlike Oblivistore or CURIOUS [85].

Pancake [36] leverages a trusted proxy to transform a set of plaintext accesses to a uniformly distributed set of encrypted accesses that can be forwarded directly to an encrypted, non-oblivious storage server. While this approach achieves high throughput, the proxy remains a bottleneck as it must maintain dynamic state about the request distribution.

Balls-into-bins analysis. Prior work derives bounds for the maximum number of balls in a bin that hold with varying definitions of high probability, but are poorly suited to our setting because they are either inefficient to evaluate or do not have a cryptographically negligible overflow probability under realistic system parameters [7, 66, 76, 77]. Berenbrink et al. [7] assume a sufficiently large number of bins to derive an overflow probability n^{-c} for n bins and some constant c (Onodera and Shibuya [69] apply this bound in the ORAM setting). Raab and Steger [77] use the first and second moment method to derive a bound where overflow probability depends on bucket load. Ramakrishna's [80] bound can be numerically evaluated but is limited by the accuracy of floating-point arithmetic, and we were unable to compute bounds with a negligible overflow probability for $\lambda \geq 44$. Reviriego et al. [83] provide an alternate formulation that can be evaluated by a symbolic computation tool, but we were unable to efficiently evaluate it with SymPy.

11 Conclusion

Snoopy is a high-throughput oblivious storage system that scales like a plaintext storage system. Through techniques that enable every system component to be distributed and parallelized while maintaining security, Snoopy overcomes the scalability bottleneck present in prior work. With 18 machines, Snoopy can scale to a throughput of 92K reqs/sec with average latency under 500ms for 2M 160-byte objects, achieving a 13.7× improvement over Obladi [26].

Acknowledgments. We thank the anonymous SOSP reviewers and our shepherd, Sebastian Angel, for their helpful feedback. We would also like to thank Hong Jun Jeon for his help in navigating the balls-into-bins literature, and Su Le, Pratyush Mishra, and students in the RISELab security group for giving feedback that improved the presentation of the paper. This work is supported by NSF CISE Expeditions Award CCF-1730628, NSF CAREER 1943347, and gifts from the Sloan Foundation, Alibaba, Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk, and VMware. This work is also supported by NSF Graduate Research Fellowships and a Microsoft Ada Lovelace Research Fellowship.

References

[1] Snoopy repository. https://github.com/ucbrise/snoopy.

[2] Trillian. https://github.com/google/trillian.

[3] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. OBLIVIATE: A data oblivious filesystem for Intel SGX. In *NDSS*, 2018.

[4] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*. USENIX, 2016.

[5] Attestation Service for Intel SGX. https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf.

[6] Kenneth E Batcher. Sorting networks and their applications. In *AFIPS*, 1968.

[7] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations: the heavily loaded case. In *STOC*, pages 745–754, 2000.

[8] 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*. ACM, 2015.

[9] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, and Srinivas Devadas. MI6: Secure enclaves in a speculative out-of-order processor. In *MICRO*. IEEE/ACM, 2019.

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

[11] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *DSN*. IEEE, 2017.

[12] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*. USENIX, 2017.

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

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

[15] T-H Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel ram. In *ASIACRYPT*. IACR, 2017.

[16] T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient oram and opram. In *ASIACRYPT*. Springer, IACR, 2017.

[17] T-H Hubert Chan and Elaine Shi. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In *TCC*. IACR, 2017.

[18] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel ram: improved efficiency and generic constructions. In *TCC*. IACR, 2016.

[19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SGXPECTRE: Stealing intel secrets from SGX enclaves via speculative execution. In *EuroS&P*. IEEE, 2019.

[20] Zitai Chen, Georgios Vasilakis, Kit Murdock, Edward Dean, David Oswald, and Flavio D Garcia. VoltPillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *USENIX Security Symposium*, 2021.

[21] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*. IEEE, 1995.

[22] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6), 1998.

[23] Robert M Corless, Gaston H Gonnet, David EG Hare, David J Jeffrey, and Donald E Knuth. On the Lambert W function. *Advances in Computational mathematics*, 1996.

[24] Manuel Costa, Lawrence Esswood, Olga Ohrimenko, Felix Schuster, and Sameer Wagh. The pyramid scheme: Oblivious RAM for trusted processors. *arXiv preprint arXiv:1712.07882*, 2017.

[25] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

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

[27] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. DORY: An encrypted search system with distributed trust. In *OSDI*. USENIX, 2020.

[28] Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Saurabh Shintre. {SEAL}: Attack mitigation for encrypted databases via adjustable leakage. In *USENIX Security Symposium*, 2020.

[29] Jack Doerner and Abhi Shelat. Scaling oram for secure computation. In *CCS*. ACM, 2017.

[30] Saba Eskandarian and Matei Zaharia. ObliDB: oblivious query processing for secure databases. *VLDB*, 2019.

[31] 5 advantages of a cloud-based EHR. https://www.carecloud.com/continuum/5-advantages-of-a-cloud-based-ehr-for-large-practices/.

[32] Christopher W Fletcher, Ling Ren, Albert Kwon, Marten Van Dijk, Emil Stefanov, Dimitrios Serpanos, and Srinivas Devadas. A low-latency, low-area hardware oblivious RAM controller. In *FCCM*. IEEE, 2015.

[33] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 1996.

[34] Michael T Goodrich. Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In *SPAA*, 2011.

[35] Google. Key transparency design doc. https://github.com/google/keytransparency/blob/master/docs/design_new.md.

[36] 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 Symposium*, 2020.

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

[38] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*, 2017.

[39] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *Security & Privacy*. IEEE, 2018.

[40] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *NSDI*. USENIX, 2016.

[41] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017.

[42] Ryan Henry. Polynomial batch codes for efficient it-pir. *PETS Symposium*, 2016.

[43] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 1990.

[44] Thang Hoang, Rouzbeh Behnia, Yeongjin Jang, and Attila A Yavuz. Mose: Practical multi-user oblivious storage via secure enclaves. In *CODASPY*. ACM, 2020.

[45] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *PETS*, 2019.

[46] Intel. Intel xeon scalable platform built for most sensitive workloads. https://www.intel.com/content/www/us/en/newsroom/news/xeon-scalable-platform-built-sensitive-workloads.html.

[47] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *STOC*, 2004.

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

[49] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking down the processor via Rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, 2017.

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

[51] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *USENIX Security Symposium*, 2021.

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

[53] Mu-Hsing Kuo. Opportunities and challenges of cloud computing to improve health care services. *Journal of medical Internet research*, 2011.

[54] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *SODA*. SIAM, 2012.

[55] Dayeol Lee, Dongha Jung, Ian T Fang, Chia-Che Tsai, and Raluca Ada Popa. An off-chip attack on hardware enclaves via the memory bus. In *USENIX Security Symposium*, 2020.

[56] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *EuroSys*. ACM, 2020.

[57] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security Symposium*, 2017.

[58] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. GhostRider: A hardware-software system for memory trace oblivious computation. *ASPLOS*, 2015.

[59] 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.

[60] Moxie Marlinspike. The difficulty of private contact discovery, January 2014. https://signal.org/blog/contact-discovery/.

[61] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback protection for trusted execution. In *USENIX Security Symposium*, 2017.

[62] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. BITE: Bitcoin lightweight client privacy using trusted execution. In *USENIX Security Symposium*, pages 783–800, 2019.

[63] Marcela S Melara, Joseph Blankstein, Aaron aind Bonneau, Edward W Felten, and Michael J Freedman. CONIKS: Bringing key transparency to end users. In *USENIX Security Symposium*, 2015.

[64] Memtier benchmark. https://github.com/RedisLabs/memtier_benchmark.

[65] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *Security & Privacy*. IEEE, 2018.

[66] Michael Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 2001.

[67] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[68] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against Intel SGX. In *Security & Privacy*. IEEE, 2020.

[69] Taku Onodera and Tetsuo Shibuya. Succinct oblivious ram. *arXiv preprint arXiv:1804.08285*, 2018.

[70] OpenEnclave. https://github.com/openenclave/openenclave.

[71] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for SGX enclaves. In *EuroSys*. ACM, 2017.

[72] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*. ACM, 1990.

[73] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. Memoir: Practical state continuity for protected modules. In *Security & Privacy*. IEEE, 2011.

[74] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor: Privacy-preserving video analytics as a cloud service. In *USENIX Security Symposium*, 2020.

[75] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Safebricks: Shielding network functions in the cloud. In *NSDI*. USENIX, 2018.

[76] Martin Raab and Angelika Steger. Balls into bins–a simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*. Springer, 1998.

[77] Martin Raab and Angelika Steger. "balls into bins"—a simple and tight analysis. In *International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 159–170. Springer, 1998.

[78] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *Security & Privacy*. IEEE, 2021.

[79] Vijaya Ramachandran and Elaine Shi. Data oblivious algorithms for multicores. *arXiv preprint arXiv:2008.00332*, 2020.

[80] MV Ramakrishna. Computing the probability of hash table/urn overflow. *Communications in Statistics-Theory and Methods*, 16(11):3343–3353, 1987.

[81] Redis. https://redis.io/.

[82] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *USENIX Security Symposium*, 2015.

[83] Pedro Reviriego, Lars Holst, and Juan Antonio Maestro. On the expected longest length probe sequence for hashing with separate chaining. *Journal of Discrete Algorithms*, 9(3):307–312, 2011.

[84] Ripple. https://ripple.com/xrp/.

[85] Cetin Sahin, Victor Zakhary, Amr El Abbadi, Huijia Lin, and Stefano Tessaro. TaoStore: Overcoming asynchronicity in oblivious data storage. In *Security & Privacy*. IEEE, 2016.

[86] Sajin Sasy, Sergey Gorbunov, and Christopher W Fletcher. ZeroTrace: Oblivious memory primitives from Intel SGX. In *NDSS*, 2018.

[87] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*. ACM, 2019.

[88] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *DIMVA*. Springer, 2017.

[89] Mary Shacklett. Financial services companies are starting to use the cloud for big data and ai processing. https://www.techrepublic.com/article/financial-services-companies-are-starting-to-use-the-cloud-for-big-data-and-ai-processing/, 2020.

[90] Solana. Solana decentralized exchange. https://soldex.ai/wp-content/uploads/2021/07/Soldex.ai-whitepaper-.pdf.

[91] Emil Stefanov and Elaine Shi. ObliviStore: High performance oblivious cloud storage. In *Security & Privacy*. IEEE, 2013.

[92] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious ram. In *NDSS*, 2012.

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

[94] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: exposing the perils of security-oblivious energy management. In *USENIX Security Symposium*, 2017.

[95] Shruti Tople, Yaoqi Jia, and Prateek Saxena. PRO-ORAM: Practical read-only oblivious RAM. In *RAID*, 2019.

[96] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security Symposium*, 2018.

[97] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *Security & Privacy*. IEEE, 2020.

[98] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy

page table-based attacks on enclaved execution. In *USENIX Security Symposium*, 2017.

[99] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *Security & Privacy*. IEEE, 2019.

[100] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. *arXiv preprint arXiv:2006.13353*, 2020.

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

[102] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security & Privacy*. IEEE, 2015.

[103] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *NSDI*. USENIX, 2017.

A Parameter analysis

Theorem 3. *For any set of R requests that are distinct and randomly distributed, number of subORAMs S, and security parameter λ, let $\mu = R/S$, $\gamma = -\log(1/(S \cdot 2^{\lambda}))$, and $W_0(\cdot)$ be branch 0 of the Lambert W function [23]. Then for the following function $f(R,S)$ that outputs a batch size, the probability that a request is dropped is negligible in λ:*

$$f(R,S) = \min\left(R,\ \mu \cdot \exp\left[W_0\left(e^{-1}(\gamma/\mu - 1)\right) + 1\right]\right).$$

Proof. Let $X_1, X_2, \ldots, X_R$ be independent 0/1 random variables that represent request i hashing to a specific subORAM where $\Pr[X_i = 1] = 1/S$. Then, $X = \sum_{i=1}^{R} X_i$ is a random variable representing the total amount of requests hashing to a specific subORAM.

We can apply the Chernoff bound here. Let $\mu = \mathbb{E}[X]$, which is $\sum_{i=1}^{R} 1/S = R/S$. Then,

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(\delta+1)^{\delta+1}}\right)^{\mu}$$

The variable X represents the total number of requests mapping to subORAM S, but we want to upper bound the number of requests received at *any* subORAM. We can define a bad event overflow that occurs when the number of requests received at any subORAM exceeds our upper bound. We can compute the probability of this bad event by taking a union bound over all S subORAMs:

$$\Pr[\text{overflow}] \leq \sum_{j=1}^{S} \Pr[X \geq (1+\delta)\mu] = S \cdot \Pr[X \geq (1+\delta)\mu]$$

In order to ensure that we do not drop a request except with negligible probability, we want $\Pr[\text{overflow}] \leq 1/2^{\lambda}$, which means we need to find some δ such that:

$$\Pr[X \geq (1+\delta)\mu] \leq \left(\frac{e^{\delta}}{(\delta+1)^{\delta+1}}\right)^{\mu} \leq \frac{1}{S \cdot 2^{\lambda}}$$

From this point, we can solve for δ to find the upper bound:

$$\begin{aligned}
-\log\left(\left(\frac{e^{\delta}}{(\delta+1)^{\delta+1}}\right)^{\mu}\right) &\geq -\log\left(\frac{1}{S \cdot 2^{\lambda}}\right) = \gamma \\
-\mu(\log(e^{\delta}) - (\delta+1)\log(\delta+1)) &\geq \gamma \\
-\delta + (\delta+1)\log(\delta+1) &\geq \frac{\gamma}{\mu} \\
(-\delta - 1) + (\delta+1)\log(\delta+1) &\geq \frac{\gamma}{\mu} - 1 \\
(\delta+1)(\log(\delta+1) - 1) &\geq \frac{\gamma}{\mu} - 1 \\
e^{\log(\delta+1)}(\log(\delta+1) - 1) &\geq \frac{\gamma}{\mu} - 1 \\
e^{\log(\delta+1)-1}(\log(\delta+1) - 1) &\geq e^{-1}\left(\frac{\gamma}{\mu} - 1\right) \\
\log(\delta+1) - 1 &\geq W_0\left(e^{-1}\left(\frac{\gamma}{\mu} - 1\right)\right) \\
\delta &\geq e^{W_0\left(e^{-1}\left(\frac{\gamma}{\mu} - 1\right)\right) + 1} - 1
\end{aligned}$$

where $W_0(\cdot)$ is branch 0 of the Lambert W function [23].

For small R, the above bound is greater than R. For $f(R,S) = R$, the overflow probability is zero, and so we can safely upper-bound f by R.

□

B Security analysis

We adopt the standard security definition for ORAM [92, 93]. Intuitively, this security definition requires that the server learns nothing about the access pattern. In the enclave setting, this means that the enclave's memory access pattern shouldn't reveal any information about the requests or data. Because Snoopy uses multiple enclaves, the communication pattern between enclaves also shouldn't reveal any information. We refer to the information that the adversary learns (the memory access patterns and communication patterns) as the "trace". At a high level, we must prove security by showing that the adversary cannot distinguish between a real experiment, where enclaves are running the Snoopy protocol on real requests and data, and an ideal experiment, where enclaves are running a simulator program that only takes as input public information. We define these experiments in detail below.

B.1 Enclave definition

We model a directed acyclic graph (DAG) of enclaves as the ideal functionality $\mathcal{F}_{\mathsf{Enc}}$ with the following interface:

- $\mathsf{E}_\mathsf{P} \leftarrow \mathsf{Load}(\mathsf{P})$: The load function takes a program P and produces an enclave DAG E_P loaded with P (the program specifies the individual programs running on each enclave and the paths of communication). This is implemented using a remote attestation procedure in Intel SGX.
- $(\mathsf{out}, \gamma) \leftarrow \mathsf{Execute}(\mathsf{E}_\mathsf{P}, \mathsf{in})$: The execute function takes an enclave DAG loaded with P, feeds in to the enclave DAG and produces the resulting output out as well as a trace of memory accesses and communication patterns between enclaves γ. Execute supports programs that communicate across enclaves and access individual enclave memories and simply outputs the trace of executing such programs.

We treat the enclave DAG as a black box that realizes the above ideal functionalities. We assume that the server cannot roll back the enclaves during execution and that Execute provides privacy and integrity for the enclave's internal memory and communication between enclaves.

Our ideal functionality interface is loosely based on the interface in ZeroTrace [86]. However, ZeroTrace only considers a single enclave whereas we consider a DAG of enclaves (similar to Opaque [103]). Also, ZeroTrace outputs proofs of correctness, whereas we use an ideal functionality where the enclave always loads and executes correctly.

B.2 Our model

We only model the case where there is a single client controlled by the adversary. We informally discuss how to extend our security guarantees to the multi-user setting in Appendix B.7.

Our ideal enclave DAG functionality hides the details of how enclaves securely communicate; using authenticated encryption and nonces to avoid replaying messages are standard techniques and discussed in other works [103]. We assume that the system configuration (the number of load balancers and subORAMs) is fixed. Also, our ideal functionality protects the contents of memory, and so we do not model the optimization (§7) where we place encrypted data in external memory in order to reduce enclave paging overhead. Finally, we do not allow the attacker to perform rollbacks attacks and we do not model fault tolerance (we do not model the system using the fault tolerance and rollback protection techniques discussed in §9).

B.3 Oblivious storage definitions

An oblivious storage scheme consists of two protocols (OStoreInitialize, OStoreBatchAccess), where OStoreInitialize initializes the memory, and OStoreBatchAccess performs a batch of accesses. We describe the syntax for both protocols below, which we will load and execute on an enclave DAG:

- OStoreInitialize(1^λ, $\mathbf{O}$), takes as input a security parameter λ and an object store $\mathbf{O}$ and runs initialization.
- $\mathbf{V} \leftarrow$ OStoreBatchAccess($\mathbf{R}$), a protocol where the client's input is a batch $\mathbf{R}$ of requests of the form (op, i, v_i) where op is the type of operation (read or write), i is an index, v_i is the value to be written (for $\mathsf{op} = \mathtt{read}$, $v_i = \perp$). The output consists of the updated secret state σ and the requested values $\mathbf{V}$ (i.e., $v_1,\ldots,v_\mu$) assigned to the $i_1,\ldots,i_\mu$ values of $\mathbf{O}$ if $\mathsf{op} = \mathtt{read}$ (for $\mathsf{op} = \mathtt{write}$, the returned value is the value before the write).

Security. The security of an oblivious storage scheme is defined using two experiments (real, ideal). In the real experiment (Figure 15), the adversary interacts with an enclave DAG loaded with the real protocol, and in the ideal experiment (Figure 16), the adversary interacts with an ideal functionality. The ideal functionality has the same interface as the real scheme but, rather than running the real protocol on the enclave DAG, it instead invokes a simulator (executing on the enclave DAG). Crucially, the simulator does not get access to the set of requests and only knows the public information, which includes the number of requests, structure of enclave DAG, and any other protocol-specific public parameters (e.g. number of load balancers and subORAMs). The

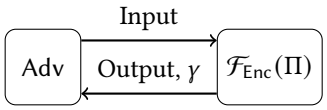


Figure 15. Real experiment for protocol Π running inside the enclave ideal functionality $\mathcal{F}_{\text{Enc}}$ where γ is the trace.

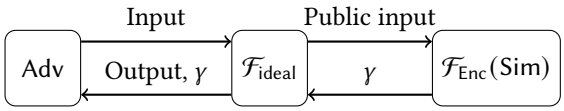


Figure 16. Ideal experiment where adversary interacts with the ideal functionality (computes the output for the given input) and the ideal functionality sends the public information to a simulator program running inside the enclave ideal functionality ($\mathcal{F}_{\text{Enc}}$) to generate the trace γ.

$bit \leftarrow \mathbf{Real}^{\text{OSTORE}}_{\Pi,\text{Adv}}(\lambda)$:

1: $(s, \mathbf{O}) \leftarrow \text{Adv}(1^\lambda)$
2: $\text{E}_\text{P} \leftarrow \mathcal{F}_{\text{Enc}}.\text{Load}(\Pi)$
3: $\gamma_0 \leftarrow \mathcal{F}_{\text{Enc}}.\text{Execute}(\text{E}_\text{P}, (\text{OSTOREINITIALIZE}, 1^\lambda, \mathbf{O}))$.
4: **for** $k = 1$ to q **do** ▷ q: polynomial #queries
5: $\quad(\mathbf{R}_k, s) \leftarrow \text{Adv}^1(\gamma_0,...,\gamma_{k-1}, \mathbf{V}_1,...,\mathbf{V}_{k-1}, s)$.
6: $\quad(\mathbf{V}_k, \gamma_k) \leftarrow \mathcal{F}_{\text{Enc}}.\text{Execute}(\text{E}_\text{P}, (\text{OSTOREBATCHACCESS}, \mathbf{R}_k))$.
7: **end for**
8: **return** $bit \leftarrow \text{Adv}(\gamma_0,...,\gamma_k, \mathbf{V}_1,...,\mathbf{V}_k, s)$.

$bit \leftarrow \mathbf{Ideal}^{\text{OSTORE}}_{\text{Sim,Adv}}(\lambda)$:

1: $(s, \mathbf{O}) \leftarrow \text{Adv}(1^\lambda)$.
2: $\text{E}_\text{P} \leftarrow \mathcal{F}_{\text{Enc}}.\text{Load}(\text{Sim})$
3: $\gamma_0 \leftarrow \text{IDEALOSTOREINITIALIZE}(\text{E}_\text{P}, 1^\lambda, \mathbf{O})$.
4: **for** $k = 1$ to q **do** ▷ q: polynomial #queries
5: $\quad(\mathbf{R}_k, s) \leftarrow \text{Adv}^1(\gamma_0,...,\gamma_{k-1}, \mathbf{V}_1,...,\mathbf{V}_{k-1}, s)$.
6: $\quad(\mathbf{V}_k, \gamma_k) \leftarrow \text{IDEALOSTOREBATCHACCESS}(\text{E}_\text{P}, \mathbf{R}_k)$.
7: **end for**
8: **return** $bit \leftarrow \text{Adv}(\gamma_0,...,\gamma_k, \mathbf{V}_1,...,\mathbf{V}_k, s)$.

[1] Security Definition 2 (weaker oblivious storage definition): Adv is not allowed to submit duplicate requests in batch R_k.

Figure 17. Real and ideal experiments for an oblivious storage scheme.

adversary can execute OSTOREINITIALIZE and a polynomial number of OSTOREBATCHACCESS for any set of requests, during which it observes the memory access patterns and communication patterns in the enclave DAG (represented by the trace produced by the Execute routine). The goal of the adversary is to distinguish between the real and ideal experiments.

An oblivious storage scheme is secure if no efficient polynomial-time adversary can distinguish between these two experiments with more than negligible probability. Our security definition has a different setup than that of traditional ORAM [92, 93] (we use a network of enclaves rather than the traditional client-server model), but our definition embodies the same security guarantees (namely, that the trace generated from an access is simulatable from public information).

We prove the security of Snoopy modularly: we first prove that our subORAM construction is secure, and then we prove that our Snoopy construction is secure when built on top of a secure subORAM. To do this, we need a slightly different notion of subORAMs. In particular, our SubORAM construction cannot be proven secure with Definition 1, since its security relies on the assumption that a batch of oblivious accesses contains *distinct* requests. In order to prove the security of our SubORAM we introduce a second, weaker security definition below.

Definition 2. (Weaker oblivious storage def.) The oblivious storage scheme Π is secure if for any non-uniform probabilistic polynomial-time (PPT) adversary Adv who *does not submit duplicated requests inside a batch* there exists a PPT Sim such that

$$\left|\Pr\left[\mathbf{Real}^{\text{OSTORE}}_{\Pi,\text{Adv}}(\lambda) = 1\right] - \Pr\left[\mathbf{Ideal}^{\text{OSTORE}}_{\text{Sim,Adv}}(\lambda) = 1\right]\right| \leq \text{negl}(\lambda)$$

$\gamma \leftarrow$ IdealOStoreInitialize(E_P, 1^λ, $\mathbf{O}$):

1: Initialize a key-value store S with contents from $\mathbf{O}$.
2: Run $\gamma \leftarrow \mathcal{F}_{\mathsf{Enc}}$.Execute($\mathsf{E}_P$,(SimOStoreInitialize, 1^λ, $|\mathbf{O}|$))
3: **return** γ.

($\mathbf{V}$, γ) $\leftarrow$ IdealOStoreBatchAccess(E_P, $\mathbf{R}$):

1: Run the batch of requests $\mathbf{R}$ on the key-value store S to produced requested values $\mathbf{V}$.
2: Run $\gamma \leftarrow \mathcal{F}_{\mathsf{Enc}}$.Execute($\mathsf{E}_P$, (SimOStoreBatchAccess, $|\mathbf{R}|$))
3: **return** ($\mathbf{V}$, γ).

Figure 18. Ideal functionalities.

where λ is the security parameter, the above experiments are defined in Figure 18 (see note 1), and the randomness is taken over the random bits used by the algorithms of Π, Sim, and Adv.

B.4 Oblivious building blocks

We use the following oblivious building blocks:

- OCmpSwap(b, x, y): If $b=1$, swap x and y.
- OCmpSet(b, x, y): If $b=1$, set $x \leftarrow y$.
- $L' \leftarrow$ OSort(L, f): Obliviously sorts the list L by some ordering function f, outputs sorted list L'.
- $L' \leftarrow$ OCompact(L, B): Obliviously compacts the list L, outputting element L_i only if $B_i=1$. The order of the original list L is preserved.

Our algorithms require only a simple "oblivious swap" primitive to build oblivious compare-and-set, oblivious sort, and oblivious compact. In our implementation, we instantiate oblivious sort using bitonic sort [6] and oblivious compaction using Goodrich's algorithm [34]. We set the client's memory to be constant size in both. OCmpSwap and OCmpSet are standard oblivious building blocks, as described in Oblix [65]. Thus, we can assume the existence of simulators SimOCmpSwap, SimOCmpSet, SimOSort, and SimOCompact. While simulator algorithms usually run in their own "address space", because we need to produce memory traces that are indistinguishable from those produced by the original algorithm, we need to pass in the address of some objects, even if the algorithms do not need to know the *values* of these objects. We define the following simulator algorithms:

- SimOCmpSwap(addr$\langle x \rangle$, addr$\langle y \rangle$)): Simulates swapping x and y given a hidden input bit.
- SimOCmpSet(addr$\langle x \rangle$), addr$\langle y \rangle$)): Simulates setting x to y given a hidden input bit.
- SimOSort(addr$\langle L \rangle$, n, f): Simulates sorting list L of length n by ordering function f.
- SimOCompact(addr$\langle L \rangle$, n, addr$\langle B \rangle$, m): Simulates compacting list L of length n using bits in list B where the number of bits in B set to 1 is m.

We additionally use OHashTable [16], which is a two-tiered oblivious hash table that consists of the polynomial-time algorithms (Construct, GetBuckets):

- $T \leftarrow$ OHashTable.Construct(D): Given some data D, output a two-tiered oblivious hash table T.
- (B_1, B_2) $\leftarrow$ OHashTable.GetBuckets(T, idx): Given an oblivious hash table T and some index idx, output pointers to the two buckets corresponding to idx. Note that these buckets may be both read from and written to.

As these algorithms are oblivious [16], we can assume the existence of a simulator SimOHashTable with algorithms (Construct, GetBuckets):

- $T \leftarrow$ SimOHashTable.Construct(addr$\langle D \rangle$, n): Given the address of data D of size n, simulate constructing an oblivious hash table.
- (B_1, B_2) $\leftarrow$ SimOHashTable.GetBuckets(T, addr$\langle$idx$\rangle$): Given a hash table T, simulate outputting pointers to two buckets corresponding to the private input idx.

Finally, we assume we have access to a keyed cryptographic hash function H.

B.5 SubORAM

We define an oblivious storage scheme SubORAM in Figure 19 that provides the interface defined in Appendix B.3 (we leave some empty lines in the protocol figure and corresponding simulator figure so that corresponding operations have the same line number).

Theorem 1. *Given a two-tiered oblivious hash table [16], an oblivious compare-and-set operator, and an oblivious compaction algorithm, the subORAM scheme described in §5 and formally defined in Figure 19 (see appendix) is secure according to Definition 2.*

Proof. We construct our simulator in Figure 20 (we leave some empty lines so that corresponding operations in Figure 19 have the same line number). We need to argue that the traces the adversary receives as a result of executing the Initialize and BatchAccess routines do not allow the adversary to distinguish between the real and ideal experiments. Communication patterns aren't a concern, as SubORAM only uses a DAG with a single enclave. Thus we only need to show that memory access patterns are indistinguishable. To simplify the proof and our description of the simulator, we assume that functions with different signatures are indistinguishable; the memory accesses of simulator functions that take fewer parameters (because they only take public input) can easily be made indistinguishable from those of the actual functions by passing in dummy arguments. We show how memory accesses are indistinguishable, first for Initialize and then for BatchAccess (line numbers correspond to Figure 19 and Figure 20).

Initialization.

- (Line 1) The subORAM algorithm takes as input an array of size n, whereas the simulator algorithm generates a random array of the same size with the same size objects. The resulting arrays are indistinguishable.
- (Line 2) These steps are the same and only involve storing the arrays that we already established are indistinguishable.

Batch access.

- (Lines 1-4) The original algorithm doesn't perform any processing while the simulator algorithm generates an array of the same size and same object size as the array passed as input to the original algorithm. Even though the objects are randomly chosen in the simulator algorithm, because the sizes of the same, both have the same memory usage.
- (Line 5) From the security of the two-tier oblivious hash table, the hash table construction algorithm and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Lines 6, 8, 9) Both use the same looping structure that depends only on public data (i.e. the number of objects and the bucket size).
- (Line 7) By the security of the oblivious hash table, the get buckets algorithm and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Lines 10, 11) By the security of the oblivious compare-and-swap, the original algorithm and the simulator algorithm produce indistinguishable memory access patterns.
- (Line 15) Both algorithms perform linear scans over an array with a public size and add an extra bit to each array entry.
- (Line 16) These lines are identical and make a new array where the size is public (same size and object size as an existing array).
- (Line 17) By the security of oblivious compaction, the original compaction algorithm and the simulator algorithm produce indistinguishable memory access patterns.

The only task that remains is to show that the responses returned in the real and ideal experiments are indistinguishable. The correctness of the results follows from Theorem 5, where we prove that our subORAM responds to read requests to an object by returning the last write to that object. □

B.6 Snoopy

We now define Snoopy as a protocol for L load balancers and S subORAMs $\mathcal{S}_1,\ldots,\mathcal{S}_S$ in Figure 21, as well as a load balancer scheme in Figure 23 and Figure 25 (we leave some empty lines in the protocol figures and corresponding simulator figures so that corresponding operations have the same line number).

Theorem 2. *Given a keyed cryptographic hash function, an oblivious compare-and-set operator, an oblivious sorting algorithm, an oblivious compaction algorithm, and an oblivious storage scheme (secure according to Definition 2), Snoopy, as described in §4 and formally defined in Figure 21 (see appendix), is secure according to Definition 1.*

Proof. Our Snoopy construction is presented in Figure 21, with the corresponding simulator in Figure 22. We again need to show that the traces that the adversary receives as a result of executing Initialize and BatchAccess do not allow the adversary to distinguish between the real and ideal experiments.

SubORAM.Initialize(1^λ,$\mathbf{O}$)

Parse $\mathbf{O}$ as $(o_1,...,o_n)$ where $o_i = (\text{idx}, \text{content})$.
Store $\mathbf{O}$.

$\mathbf{V} \leftarrow$ SubORAM.BatchAccess($\mathbf{R}$)

Parse $\mathbf{R}$ as $(r_1,...,r_N)$, where $r_i = (\text{type}, \text{idx}, \text{content})$.
if $\mathbf{R}$ contains duplicates **then**
    return $\perp$.
end if
Set $T \leftarrow$ OHashTable.Construct($\mathbf{R}$).
for $i = 1,...,n$ **do**
    Set $\mathbf{Bkt}_1, \mathbf{Bkt}_2 \leftarrow$ OHashTable.GetBuckets(T, $\mathbf{O}[i]$.idx).
    for $j = 1,2$ **do**
        for $l = 1,...,|\mathbf{Bkt}_j|$ **do**
            OCmpSet(($\mathbf{Bkt}_j[l]$.idx $\stackrel{?}{=}$ $\mathbf{O}[i]$.idx), $\mathbf{O}[i]$.content, $\mathbf{Bkt}_j[l]$.content).
            OCmpSet(($\mathbf{Bkt}_j[l]$.idx $\stackrel{?}{=}$ $\mathbf{O}[i]$.idx) $\wedge$ ($\mathbf{Bkt}_j[l]$.type $\stackrel{?}{=}$ write), $\mathbf{Bkt}_j[l]$.content, $\mathbf{O}[i]$.content).
        end for
    end for
end for
Scan through T, marking each entry i with bit $b_i = 0$ if it is a dummy, setting $b_i = 1$ otherwise.
Set $\mathbf{B} \leftarrow (b_1,...,b_{|T|})$.
Run $\mathbf{V} \leftarrow$ OCompact(T, $\mathbf{B}$).
return $\mathbf{V}$.

Figure 19. Our subORAM construction.

SimSubORAM.Initialize(1^λ,$|\mathbf{O}|$)

Let $(n,\kappa) = |\mathbf{O}|$ (κ is the object size). Create an array $\mathbf{O} = o_1,...,o_n$ of random entries of size κ, where $o_i = (\text{idx}, \text{content})$.
Store $\mathbf{O}$.

SimSubORAM.BatchAccess(N)

Let N be a public parameter, which denotes the number of requests that the input batch contains.
Choose N random distinct identifiers $\text{idx}_1,...,\text{idx}_N$ where for all $i \in [N]$, idx_i is an idx value in $\mathbf{O}$.
Create $\mathbf{R}$ of the form $(r_1,...,r_N)$, where $r_i = (\text{read}, \text{idx}_i, \perp)$.

Run $T \leftarrow$ SimOHashTable.Construct(addr$\langle\mathbf{R}\rangle$,N).
for $i = 1,...,n$ **do**
    Set $\mathbf{Bkt}_1, \mathbf{Bkt}_2 \leftarrow$ SimOHashTable.GetBuckets(T,addr$\langle\mathbf{O}[i]$.idx$\rangle$).
    for $j = 1,2$ **do**
        for $l = 1,...,|\mathbf{Bkt}_j|$ **do**
            SimOCmpSet(addr$\langle\mathbf{O}[i]$.content$\rangle$, addr$\langle\mathbf{Bkt}_j[l]$.content$\rangle$).
            SimOCmpSet(addr$\langle\mathbf{Bkt}_j[l]$.content$\rangle$, addr$\langle\mathbf{O}[i]$.content$\rangle$).
        end for
    end for
end for
Scan through T, marking each entry with bit $b_i = 0$.
Set $\mathbf{B} \leftarrow (b_1,...,b_{|T|})$.
Run $\mathbf{V} \leftarrow$ SimOCompact(addr$\langle T\rangle$, $|T|$, addr$\langle\mathbf{B}\rangle$, N).

Figure 20. Simulator algorithms SimSubORAM = (Initialize, BatchAccess).

The communication patterns in the real and ideal experiments are indistinguishable. Both experiments perform setup at the first load balancer and then copy state to the remaining load balancer (communication pattern is deterministic). For BatchAccess, in both experiments, we choose a random load balancer, which then communicates with every subORAM (the amount of data sent to each subORAM depends only on public information). Thus there is no difference in the distribution of communication patterns between the real and ideal experiments.

We now discuss memory access patterns. As in the proof for Theorem 2, to simplify the proof and our description of the simulator, we assume that functions with different signatures are indistinguishable; the memory accesses of simulator functions that take fewer parameters (because they only take public input) can easily be made indistinguishable from those of the actual functions by passing in dummy arguments. As is clear from Figure 21 and Figure 22, the Initialize and BatchAccess algorithms are identical except that (1) the simulator algorithm generates random objects and random requests rather than taking them as input, and (2) the simulator algorithm calls the SimLoadBalancer algorithms. Thus the only task that remains is to show that the memory access patterns generated by the LoadBalancer and SimLoadBalancer algorithms are indistinguishable.

We start with Initialize and then examine BatchAccess (line numbers correspond to Figure 23, Figure 24, Figure 25, Figure 26).

Initialization.

- (Lines 1-2) The load balancer algorithm takes an array $\mathbf{O}$ whereas the simulator algorithm generates a random array of the same size (same number of objects and same object size). Thus the memory used by these arrays is indistinguishable.
- (Lines 3-8) These lines are identical. We sample a key and then perform a linear scan over an array where the size of the array and object size is public, attaching a tag to each element.
- (Line 9) By the security of our oblivious sort, the sorting algorithms over different arrays with the same length, same object size, and same ordering function produce indistinguishable memory access patterns because of the existence of the simulator function that only takes in array length, object size, and the ordering function.
- (Lines 10-17) These lines are identical. We iterate over the array where the array size is public. We write the algorithm as branching based on a comparison to private data in order to improve readability, but this would in practice be implemented using OCmpSet in the original algorithm and SimOCmpSet in the simulator algorithm, which produce indistinguishable access patterns.
- (Lines 19-21) By the security of the underlying subORAM scheme, the initialize procedure for the subORAM and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Lines 22-23) These lines are identical and only store a cryptographic key.

Batch access.

- (Lines 1-2) Establishing parameters and hash functions.
- (Lines 3-4) The load balancer receives a list of requests whereas the simulator algorithm generates a random array of the same size (same number of requests and same format). Thus the memory used by these arrays is indistinguishable.
- (Lines 5-11) These lines are identical and only compute a function based on public information and perform a linear scan over an array (same size and format in both). Thus the memory access patterns are indistinguishable.
- (Line 12) By the security of the oblivious sorting algorithm, the oblivious sort and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Line 13) These lines are identical and require accessing α objects in a fixed location where α is computed using public information.
- (Line 14) By the security of the oblivious compaction algorithm, the oblivious compaction and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Lines 15-17) By the security of the underlying subORAM scheme, the batch access algorithm and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Line 18) These lines are identical and create an array where the number of objects is based on public information and the object size is a public parameter.
- (Line 19) These lines set the same function.
- (Line 20) By the security of the underlying sorting algorithm, the oblivious sort and the corresponding simulator algorithm produce indistinguishable memory access patterns.
- (Line 21-24) The structure of the loop is the same in both algorithms and depends only on public information ($N+\alpha S$), and the compare-swap primitive guarantees that the algorithm and the simulator algorithm produce indistinguishable memory access patterns.

$\mathsf{Snoopy.Initialize}_{L,S}(1^\lambda, \mathbf{O})$

1: Let L be a public parameter, which denotes the number of load balancers.
2: Let S be a public parameter, which denotes the number of used SubORAMs.
3: $k \leftarrow \mathsf{LoadBalancer.Initialize}_S(1^\lambda, \mathbf{O})$.
4: Send k to the remaining $L-1$ load balancers.

$\mathbf{V} \leftarrow \mathsf{Snoopy.BatchAccess}_{L,S}(\mathbf{R})$

1: Let L be a public parameter, which denotes the number of load balancers.
2: Let S be a public parameter, which denotes the number of used SubORAMs.
3: Wait to receive $|\mathbf{R}|$ requests.
4: Pick at random a load balancer i.
5: Run $\mathbf{V}_i \leftarrow \mathsf{LoadBalancer}_i.\mathsf{BatchAccess}_S(\mathbf{R})$.
6: **return** $\mathbf{V}_i$.

Figure 21. Our Snoopy construction.

$\mathsf{SimSnoopy.Initialize}_{L,S}(1^\lambda, |\mathbf{O}|)$

1: Let L be a public parameter, which denotes the number of load balancers.
2: Let S be a public parameter, which denotes the number of used SubORAMs.
3: $k \leftarrow \mathsf{SimLoadBalancer.Initialize}_S(1^\lambda, |\mathbf{O}|)$.
4: Send k to the remaining $L-1$ load balancers.

$\mathsf{SimSnoopy.BatchAccess}_{L,S}(N)$

1: Let L be a public parameter, which denotes the number of load balancers.
2: Let S be a public parameter, which denotes the number of used SubORAMs.
3: Let N be the number of requests.
4: Pick at random a load balancer i.
5: Run $\mathsf{SimLoadBalancer}_i.\mathsf{BatchAccess}_S(N)$.
6:

Figure 22. Simulator algorithms SimSnoopy = (Initialize, BatchAccess).

- (Line 25) Creates a list where the list size is based on public information ($N+\alpha S$) and the object size is public.
- (Line 26) By the security of the underlying compaction algorithm, the oblivious compaction and the corresponding simulator algorithm produce indistinguishable memory access patterns.

While the memory access patterns generated are indistinguishable in all cases, the adversary could potentially be able to distinguish between the real and ideal experiments if the adversary could cause the responses between the real and ideal experiments to differ. The only way that the adversary could do this is if the number of requests assigned to a subORAM exceeds $f(N,S)$ for N total requests and S subORAMs. The load balancer algorithm guarantees that requests in a batch are distinct (we use oblivious compaction to remove duplicates) and randomly distributed (we use a keyed hash function). Furthermore, the attacker cannot learn information about how requests are routed to subORAMs because the access patterns do not leak the assignment of requests to subORAMs (as proven above). Thus we can apply Theorem 3, and so the probability that a batch overflows is negligible in λ. Finally, Theorem 4 guarantees that reads always see the result of the last write, and so, the probability that the adversary can distinguish between the real experiment and the ideal experiment is negligible in λ. □

B.7 Discussion of multiple clients

Our proof only considers a single client, and so we briefly (and informally) discuss how to extend our guarantees to multiple clients. In the case where multiple clients are controlled by a single adversary, we simply need to modify the adversary to choose requests for each client, and then the clients forward the requests to the oblivious storage system. The oblivious storage protocol and the ideal functionality then needs to route the correct response to the correct client (rather than sorting by object ID on line 19 in Figure 25, the load balancer can sort by the client ID, object ID, bit b tuple).

We now consider the case where there is an honest client submitting read requests and all other clients are controlled by the adversary. Note that write requests cannot be private in the case where the adversary can make read requests, as the adversary

$k \leftarrow$ LoadBalancer.Initialize$_S(1^\lambda, \mathbf{O})$

1: Parse $\mathbf{O}$ as $o_1,...,o_n$.
2: Let S be a public parameter, which denotes the number of used SubORAMs.
3: Let H be a keyed cryptographic hash function that outputs an element in $[S]$.
4: Sample a secret key $k \xleftarrow{R} \{0,1\}^\lambda$.
5: **for** $i=1,...,n$ **do**
6: Attach to o_i the tag $t = H_k(o_i.\text{idx})$.
7: **end for**
8: Let f_{order} be the ordering function that orders by tag t.
9: $\mathbf{O} \leftarrow \text{OSort}(\mathbf{O}, f_{\text{order}})$.
10: Let $x \leftarrow 0$.
11: Let prev $\leftarrow \perp$.
12: **for** $i=1,...,|\mathbf{O}|$ **do**
13: **if** $\mathbf{O}[i].t \neq$ prev **then**
14: Let $y_x \leftarrow i$.
15: Let $x \leftarrow x+1$.
16: Let prev $\leftarrow \mathbf{O}[i].t$.
17: **end if**
18: **end for**
19: **for** $i=1,...,S$ **do**
20: Run SubORAM.Initialize$(1^\lambda, \mathbf{O}[y_{i-1}:y_i])$.
21: **end for**
22: Store k.
23: **return** k.

Figure 23. Our load balancer initialization construction. Lines 13-16 would in practice be implemented using OCmpSet, but we write it using an if statement that depends on private data to improve readability.

can always read all objects to tell what objects was written to by the honest client. We simply want to hide the contents of the read requests made by the honest client (we do not hide the timing or the number). In our proof, we show that the trace generated by operating on the batch of requests submitted by the adversary is indistinguishable from the trace generated by operating on a random batch of requests, and so the execution trace will not reveal information about the honest client's accesses. Using the modification described above, we also ensure that the correct responses are routed to the correct client, and so the adversary cannot learn information about the honest client's read requests from the returned responses.

C Linearizability

Snoopy implements a *linearizable key-value store*. We define the following terms:

- An operation o has both a start time o_{start} (the time at which the operation was received by a load balancer), and an end time o_{end} (the time at which the operation was committed by the load balancer).
- Operation o' follows operation o in real-time ($o \underset{rt}{\longrightarrow} o'$) if $o_{end} < o'_{start}$.
- o' and o are said to be concurrent if neither o nor o' follow each other.
- Operations can be either reads ($read(x)$, which reads key x), or writes ($write(x,v)$, which writes value v to key x).

Linearizability requires that for any set of operations, there exists a total ordered sequence of these operations (a linearization – we write $o \rightarrow o'$ if o' follows o in the linearization) such that:

- The linearization respects the real-time order of operations in the set: If $o \underset{rt}{\longrightarrow} o'$ then $o \rightarrow o'$ **(C1)**.
- The linearization respects the sequential semantics of the underlying data-structure. Snoopy follows the semantics of a hashmap: given two operations o and o' on the same key, where o is a write $write(x,v)$, and o' is a read $read(x)$, then, if there does not exist an o'' such that $o'' = write(x,v')$ and $o \underset{rt}{\longrightarrow} o'' \underset{rt}{\longrightarrow} o'$, then $read(x) = v$. In other words, the data structure always returns the value of the latest write to that key **(C2)**.

As in our security proofs, we prove linearizability separately for our subORAM scheme and for Snoopy instantiated with any subORAM.

$k \leftarrow$ SimLoadBalancer.Initialize(1^λ, $|\mathbf{O}|$)

1: Let $(n,\kappa) = |\mathbf{O}|$. ▷ κ is the size of the object
2: Create an array $\mathbf{O}$ $(1,o_1),(2,o_2),...,(n,o_n)$ of the form (idx, content), where o_i is a random entry of size κ.
3: Let H be a keyed cryptographic hash function that outputs an element in $[S]$.
4: Sample a secret key $k \xleftarrow{R} \{0,1\}^\lambda$.
5: **for** $i = 1,...,n$ **do**
6: Attach to o_i the tag $t = H_k(o_i.\text{idx})$.
7: **end for**
8: Let f_{order} be the ordering function that orders by tag t.
9: OSort($\mathbf{O}$, f_{order}).
10: Let $x \leftarrow 0$.
11: Let prev $\leftarrow \perp$.
12: **for** $i = 1,...,|\mathbf{O}|$ **do**
13: **if** $\mathbf{M}[i].t \neq$ prev **then**
14: Let $y_x \leftarrow i$.
15: Let $x \leftarrow x+1$.
16: Let prev $\leftarrow \mathbf{O}[i].t$.
17: **end if**
18: **end for**
19: **for** $i = 1,...,S$ **do**
20: Run SimSubORAM$_i$.Initialize(1^λ, $|\mathbf{O}[y_{i-1} : y_i]|$).
21: **end for**
22: Store k.
23: **return** k.

Figure 24. Load balancer simulator for SimLoadBalancer.Initialize. Lines 13-16 would in practice be implemented using OCmpSet, but we write it using an if statement that depends on private data to improve readability.

Theorem 4. *Snoopy is linearizable when the subORAM is instantiated with a oblivious storage scheme that is secure according to Definition 2.*

Proof. We prove that there exists a linearization that follows the hashmap's sequential specification: each operation is totally ordered according to the (batch commit time *epoch*, load balancer id *lb*, operation type *optype*, batch insertion index *ind*) tuple (sorting first by batch commit time, next by load balancer id, next giving priority to reads over writes, and finally by arrival order) . Let $o_1 \rightarrow o_2 \rightarrow ... \rightarrow ..o_n$ be the resulting linearization. We prove the aforementioned statement in two steps: (1) the statement holds true for $o_n \rightarrow o_{n+1}$, and (2) the statement holds true transitively. Note that we assume load balancers and subORAMs can take a single action per timestep.

1. $o_n \rightarrow o_{n+1}$ We prove this by contradiction. Assume that $o \rightarrow o'$ violates either condition C1 or condition C2.
   - **(C1)** Assume that condition C1 is violated: $o_{end} \geq o'_{start}$. Now, consider $o \rightarrow o'$: it follows by assumption that $(batch_o, lb_o) \leq (batch_{o'}, lb_{o'})$. If $lb_o == lb_{o'}$, o and o' are either in the same epoch or o' is in the epoch that follows o at the same load balancer. In both cases, o' cannot have a start time greater or equal than o's start time: each load balancer processes each epoch sequentially and waits for all batches to commit. We have a contradiction. Consider next the case in which $batch_o == batch_{o'}$ and $lb_o \leq lb_{o'}$. We have $o_{start} < batch_o < o_{end}$ and $o'_{start} < batch_{o'} < o'_{end}$. As $batch_o == batch_{o'}$, we have $o'_{start} < epoch_o < o_{end}$. We once again have a contradiction.
   - **(C2)** Assume that condition C2 is violated: $o = write(x,v)$ and $o' = read(x)$, but o' returns $v \neq v'$ and there does not exist an o' such that $o \xrightarrow[rt]{} o'' \xrightarrow[rt]{} o'$. We consider two cases: (1) o and o' are in different batches, and (2) o and o' are in the same batch. First, consider the case in which o and o' are in different batches and $batch_o < batch_{o'}$ (if o and o' write to the same key x and are in different batches, then $batch_o \neq batch_{o'}$ as subORAMs processes batches of requests sequentially). It follows that o' executed after o. There are two cases: (1) o is the write in the batch with the highest index, and (2) there exists a write o'' with a higher index. In the latter case, we have a contradiction: our linearization order orders writes by index, as such there exists an intermediate write o'' in the linearization order $o \rightarrow o'' \rightarrow o'$. Instead, consider o to be the write with the highest index. This write gets persisted to the subORAM as part of the batch. By the correctness of the underlying oblivious storage scheme, a read from oblivious storage (instantiated

$\mathbf{V} \leftarrow \mathsf{LoadBalancer.BatchAccess}_S(\mathbf{R})$

Let S be a public parameter, which denotes the number of used SubORAMs.
Let $\mathsf{H}_k(\cdot)$ be a cryptographic hash function keyed by stored key k that outputs an element in $[S]$.
Parse $\mathbf{R}$ as $(r_1,...,r_N)$, where $r_i = (\mathsf{type}, \mathsf{idx}, \mathsf{content})$.

Compute $\alpha \leftarrow f(N,S)$ and initialize the empty list $\mathbf{L}$ of size $N+\alpha S$.
for $i = 1,...,N$ **do**
    $\mathbf{L}[i] = (r_i.\mathsf{type}, r_i.\mathsf{idx}, r_i.\mathsf{content}, \mathsf{H}_k(r_i.\mathsf{idx}))$.
end for
$\mathbf{L}' \leftarrow$ Create a copy of $\mathbf{L}$.
Append to $\mathbf{L}'$ α dummy requests for each SubORAM of the form $(\mathsf{read}, \mathsf{idx}, \perp, s)$, where idx is $\mathsf{H}_k(\mathsf{idx}) = s$.
Let f_{order} be the ordering function that orders by SubORAM and then by type (where $\perp$ is last and treated as read).
Run $\mathbf{L}' \leftarrow \mathsf{OSort}(\mathbf{L}, f_{\mathsf{order}})$.
Tag the first α distinct requests per SubORAM with $b = 1$ and the remaining requests with $b = 0$.
Set $\mathbf{B} \leftarrow (b_1,...,b_{N+\alpha S})$ and run $\mathbf{L}' \leftarrow \mathsf{OCompact}(\mathbf{L}', \mathbf{B})$.
for $i = 1,...,S$ **do**
    Run $\mathbf{V}_i \leftarrow \mathsf{SubORAM}_i.\mathsf{BatchAccess}(\mathbf{L}'[(i-1)\alpha+1 : i\alpha])$.
end for
Set $\mathbf{X} \leftarrow (\mathbf{V}_1,...,\mathbf{V}_S,\mathbf{L})$ tagging all responses with $b = 0$ and requests with $b = 1$.
Let f_{order} be the ordering function that orders by idx and then by b (i.e., giving priority to responses over requests).
Set $\mathbf{X}' \leftarrow \mathsf{OSort}(\mathbf{X}, f_{\mathsf{order}})$.
Set $\mathsf{prev} \leftarrow \perp$.
for $i = 1,...,|X'|$ **do**
    $\mathsf{OCmpSet}(b_i \stackrel{?}{=} 0, \mathsf{prev}, \mathbf{X}'[i].\mathsf{content})$ and $\mathsf{OCmpSet}(b_i \stackrel{?}{=} 1, \mathbf{X}'[i].\mathsf{content}, \mathsf{prev})$.
end for
Set $\mathbf{B} \leftarrow (b_1,...,b_{N+\alpha S})$.
Run $\mathbf{V} \leftarrow \mathsf{OCompact}(\mathbf{X}', \mathbf{B})$.
return $\mathbf{V}$.

Figure 25. Our load balancer construction.

in our system as a subORAM, see Theorem 5) returns the latest write to that key. As such, if o' reads x in a batch that follows o's write to x with no intermediate writes to that key, o' will return the value written by o. We have a contradiction once again. (2) If o and o' are instead in the same batch, then $batch_o == batch'_o$. By our linearization order specification, reads are always ordered before writes in a batch, so $o' \rightarrow o$. We have a contradiction.

2. Transitivity. The proof holds trivially for chains of arbitrary length $o_1 \rightarrow .. \rightarrow o_n$ due the transitive nature of inequalities and the pairwise nature of operation correctness on a hashmap.

□

Theorem 5. *Our subORAM (Figure 19) always returns the value of the latest write to an object, provided that it is instantiated from a two-tiered oblivious hash table [16], an oblivious compare-and-set operator, and an oblivious compaction algorithm.*

Proof. We prove this by contradiction. Assume that the last write to object o was value v and a subsequent read of object o in epoch i returns value v' where $v \neq v'$. Because reads are ordered before writes in the same epoch, a write cannot take place between the end of the end of epoch $i-1$ and a read in epoch i. Then, by the correctness of the oblivious hash table (which we use to retrieve the correct request for an object when scanning through all objects), the oblivious compare-and-set primitive (which copies the object value correctly to the request's response data if the request is a read), and oblivious compaction (which ensures that entries in the hash table corresponding to real requests are returned) it must be the case that the value for object o in the subORAM at the end of epoch $i-1$ is v'. By the correctness of our oblivious hash table (which we use to retrieve the correct request for an object when scanning through all objects) and oblivious compare-and-set primitive (which copies the request value correctly to the object value if the request is a write) and because write requests in the same batch are distinct (our load balancer deduplicates requests in the same epoch), the last write to object o before epoch i must have been value v'. Thus we have reached a contradiction ($v \neq v'$), completing the proof. □

SimLoadBalancer.BatchAccess(N)

Let N be a public parameter, which denotes the number of requests that the queried batch contains. Let S be a public parameter, which denotes the number of used SubORAMs.
Let $\mathsf{H}_k(\cdot)$ be a cryptographic hash function keyed by stored key k that outputs an element in $[S]$.
Choose N random identifiers idx_1,...,idx_N where for all $i \in [N]$, idx_i is an idx value in $\mathbf{O}$.
Create $\mathbf{R}$ of the form $(r_1, ..., r_N)$, where $r_i = (\mathsf{read}, \mathsf{idx}_i, \perp)$.
Compute $\alpha \leftarrow f(N,S,\lambda)$ and initialize the empty list $\mathbf{L}$ of size $N+\alpha S$.
for $i = 1,...,N$ **do**
    $\mathbf{L}[i] = (r_i.\mathsf{type}, r_i.\mathsf{idx}, r_i.\mathsf{content}, \mathsf{H}_k(r_i.\mathsf{idx}))$.
end for
$\mathbf{L}' \leftarrow$ Create a copy of $\mathbf{L}$.
Append to $\mathbf{L}'$ α dummy requests for each SubORAM of the form $(\mathsf{read}, \mathsf{idx}, \perp, s)$, where idx is $\mathsf{H}_k(\mathsf{idx}) = s$.
Let f_{order} be the ordering function that orders by SubORAM and then by type (where $\perp$ is last and treated as read).
Run $\mathsf{SimOSort}(\mathsf{addr}\langle\mathbf{L}'\rangle, |\mathbf{L}'|, f_{\mathsf{order}})$.
Tag the first α requests per SubORAM with $b = 1$ and the remaining requests with $b = 0$.
Set $\mathbf{B} \leftarrow (b_1,...,b_{N+\alpha S})$ and run $\mathsf{SimOCompact}(\mathsf{addr}\langle\mathbf{L}',\rangle, N+\alpha S, \mathsf{addr}\langle\mathbf{B}\rangle, \alpha S)$.
for $i = 1,...,S$ **do**
    Run $\mathbf{V}_i \leftarrow \mathsf{SimSubORAM}_i.\mathsf{BatchAccess}(\alpha)$.
end for
Let $\mathbf{X}$ be an array of $N+\alpha S$ objects the same size as the objects in $\mathbf{L}$ with a tag bit.
Let f_{order} be the ordering function that orders by idx and then by b (i.e., giving priority to responses over requests).
Run $\mathsf{SimOSort}(\mathsf{addr}\langle\mathbf{X}\rangle, |\mathbf{X}|, f_{\mathsf{order}})$.
Set $\mathsf{prev} \leftarrow \perp$.
for $i = 1,...,|\mathbf{X}'|$ **do**
    $\mathsf{SimOCmpSet}(\mathsf{addr}\langle\mathsf{prev}\rangle, \mathsf{addr}\langle\mathbf{X}'[i].\mathsf{content}\rangle)$ and $\mathsf{SimOCmpSet}(\mathsf{addr}\langle\mathbf{X}'[i].\mathsf{content}\rangle, \mathsf{addr}\langle\mathsf{prev}\rangle)$.
end for
Set $\mathbf{B} \leftarrow (b_1,...,b_{N+\alpha S})$.
Run $\mathsf{SimOCompact}(\mathsf{addr}\langle\mathbf{X}'\rangle, |\mathbf{X}'|, \mathsf{addr}\langle\mathbf{B}\rangle, N)$.

Figure 26. Load balancer simulator for SimLoadBalancer.BatchAccess.

D Access control

Throughout the paper, we assume that all clients are trusted to make any requests for any objects. However, practical applications may require access control. We now (informally) describe how to implement access control for Snoopy. A plaintext system can store an access control matrix and, upon receiving a request, look up the user ID and object ID in the matrix to check if that user has the privileges to make that request. In an oblivious system, the challenge is that the load balancer cannot query the access control matrix directly, as the location in the access control matrix reveals the object ID requested by the client. We instead need to access the access control matrix obliviously.

We can do this using Snoopy recursively. In addition to the objects themselves, the subORAMs now need to store the access control matrix, where each object has the tuple (user ID, object ID, type) as the key (where type is either "read" or "write") and 1 or 0 as the value depending on whether or not the user has permission for that operation. The load balancer then needs to obliviously retrieve the access-control rule pertaining to the requests it received from the clients and apply the access-control rule when generating responses for the clients. Notably, if a client does not have permission to perform a read, Snoopy should return a null value instead of the object value, and if the client does not have permission to perform a write, it should not copy the value from the request to the object. In order to ensure that a user is querying with the correct user ID, users should authenticate to the load balancer using a standard authentication mechanism (e.g. password or digital signature).

Now, upon receiving a request, the load balancer generates a read request to the access control matrix for the tuple (user ID, object ID, type) corresponding to the original request. The load balancer generates batches of access-control read requests that it shards across the subORAMs. This is equivalent to running Snoopy recursively where the load balancer acts as both a client and load balancer for the batch of access-control read requests. When the load balancer receives the results of the access-control read requests, it then matches the access-control responses to the original requests by performing an oblivious sort by (user ID, object ID, type) on both the access-control responses and the original list of requests. The load balancer scans

through the lists in tandem (examine both lists at index 0, then at index 1, etc.), copying the bit b returned in the access-control response to the original request. The load balancer then sends the original requests (including this new bit b) to the subORAMs as in the original design of Snoopy.

When executing the requests, the subORAMs additionally check the value of b in the oblivious compare-and-set operation (lines 10 and 11 in Figure 19) to ensure that the operation is permitted before performing it. Note that it is critical that we hide which operations are permitted and which are not during execution; otherwise, an attacker can submit requests that aren't permitted and, by observing execution, see where in the sorted list of requests the failed request was (which leaks information about the permitted requests). Executing requests with access control now requires two epochs of execution (one to query the access control matrix and one to process the client's actual request) to return the response to the user.