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.

CCS Concepts: • **Security and privacy** → **Database and storage security**.

Keywords: Oblivious RAM, Scalability

1 Introduction

Organizations increasingly outsource sensitive data to the cloud for better convenience, cost-efficiency and availability [32, 54, 90]. Encryption cannot fully protect this data: how the user accesses data (the "access pattern") can leak sensitive information to the cloud [13, 29, 38, 49, 51, 53]. 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, 34, 73, 83, 86, 92–94, 102]. 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 [66, 83, 94], scale to large data sizes via data parallelism [60], and improve request throughput [26, 86, 102]. 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, 38, 49, 51, 53]. 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 [83, 93, 94].

Thus high-throughput oblivious storage systems are all built on hierarchical [93] or tree-like [83, 94] structures and either require a centralized coordination point (e.g., a query log [14, 102] or trusted proxy [8, 26, 86, 92]) 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?*

[*] Equal contribution.

SOSP '21, October 26–29, 2021, Virtual Event, Germany
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483562

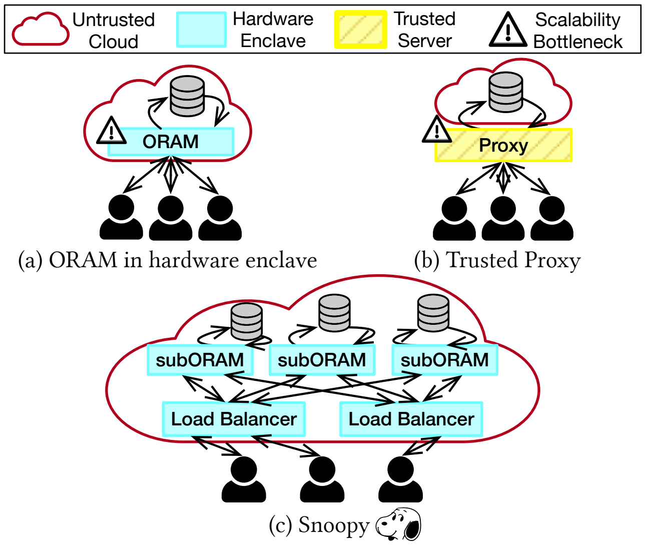
Removing the scalability bottleneck. In this work, we propose Snoopy (scalable nodes for oblivious object repository), a high-throughput oblivious storage system that scales similarly to a plaintext storage system. While our system is secure for any workload, we design it for high-throughput workloads. 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, 66, 87]. 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, 42, 56, 58, 68, 89, 99, 103]. 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, 66, 87] 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.



(a) ORAM in hardware enclave

(b) Trusted Proxy

(c) Snoopy

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.

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 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, 66, 87] 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, 66, 87], 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 [83, 94]. 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 [86], or PrivateFS [102]. For large data sizes and low request volume, a system like Shroud [60] 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, 57]; 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, 42, 68, 89], branch prediction [58], paging-based attacks [99, 103], and memory bus snooping [56]. By using oblivious algorithms, Snoopy defends against this class of attacks. Snoopy does not defend against enclave integrity attacks such as rollback [74] and transient execution attacks [19, 79, 88, 97, 98, 100, 101], 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 [39, 59]).

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 [74]. 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. 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, 69, 95] or denial-of-service attacks due to memory corruptions [40, 50]. We additionally consider transient execution attacks [19, 79, 88, 97, 98, 100, 101] 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 [57], 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 the full version [27]). 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 [44]: 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 the full version [27].

2.1 Formalizing security

We formalize our system and prove its security in the full version [27]. 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 [34]). 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 the full version [27].

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.

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

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

where λ is the security parameter, the real and ideal experiments are defined informally above and formally in the full version [27], 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 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 is secure according to Definition 1 where the adversary can only submit unique requests.*

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, Snoopy, as described in §4, 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 the full version [27].

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 [83, 94]. A central proxy running on a lightweight, trusted machine keeps a mapping of objects to servers. The client sends its request to the proxy, which then

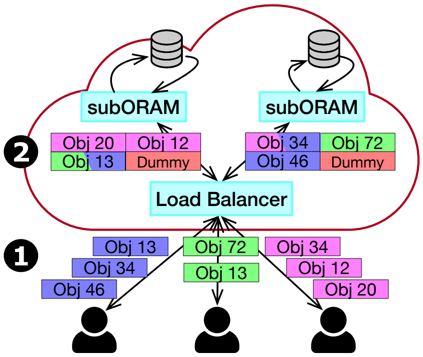


Figure 2. Secure distribution of requests in Snoopy. ❶ The load balancer receives requests from clients. ❷ At the end of the epoch, the load balancer generates a batch of requests for each subORAM, padding with dummy requests as necessary.

accesses the server currently storing that object and remaps that object to a new server [8, 92]. 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, 66, 87] 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 [61]. Privacy-preserving cryptocurrency light clients can also benefit from Snoopy. These allow lightweight clients to query full nodes for relevant transactions [63]. 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 [85, 91], 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, 64] 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 [36].

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 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

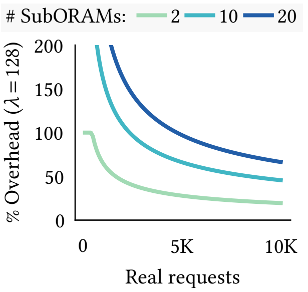


Figure 3. Dummy request overhead. A 50% overhead means for every two real requests there is one dummy request.

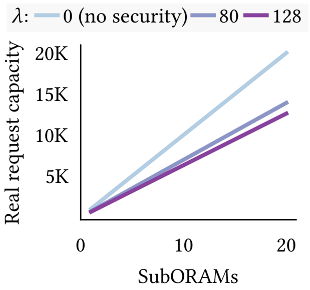


Figure 4. The total real request capacity of our system for an epoch, assuming ≤1K requests per subORAM per epoch.

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 [77]: 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 the full version [27]:

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:*

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

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, 67, 77, 78].

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 [35]. 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.

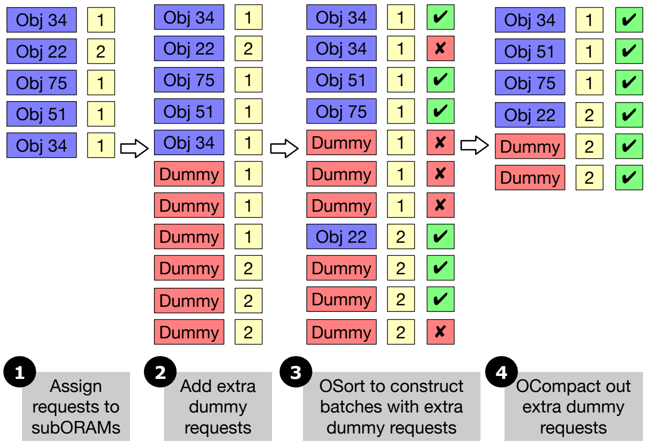


Figure 5. Generating batches of requests at the load balancer.

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.

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

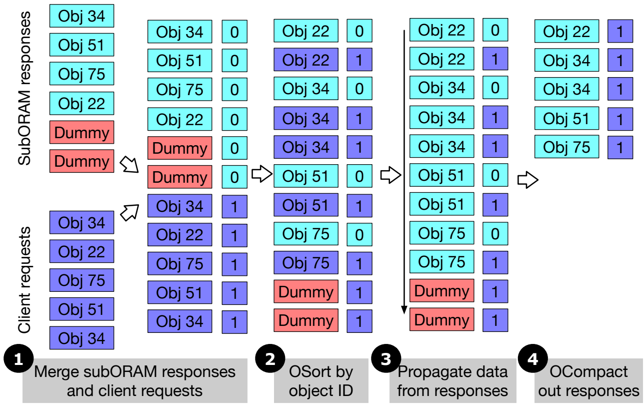

SubORAM responses
Client requests
❶ Merge subORAM responses and client requests
❷ OSort by object ID
❸ Propagate data from responses
❹ OCompact out responses

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

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 $b = 1$ (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, 86, 92]. 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 [28, 30]. 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 [61]. 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, 55].

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 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

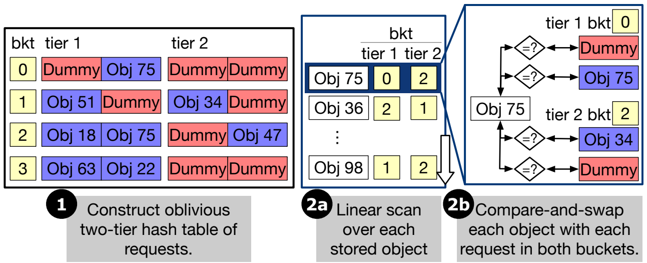


Figure 7. Processing a batch of requests at a subORAM.

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 $L_{\mathsf{LB}}(R,S)$ 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:

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

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:

$$L_{\mathsf{Sys}} \leq 5T/2 \tag{2}$$

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}:

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

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 [71] and Intel SGX v2.13. We use gRPC v1.35 for communication and OpenSSL for cryptographic operations. Our bitonic sort [6] and oblivious compaction [35] 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 [72]. 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 [75, 76].

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)

	Redis [82]	Obladi [26]	Oblix [66]	**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).

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 [82] 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 [66]).

8.1 Baselines

We compare Snoopy to three state-of-the-art baselines: Obladi [26] is a batched, high-throughput oblivious storage system, Oblix [66] efficiently leverages enclaves for oblivious storage, and Redis [82] 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 [83] 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 [66] 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 [66]).

Redis. To measure the overhead of security (obliviousness), we compare Snoopy to an insecure baseline Redis [82], 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 [65], 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 $\leq$ 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).

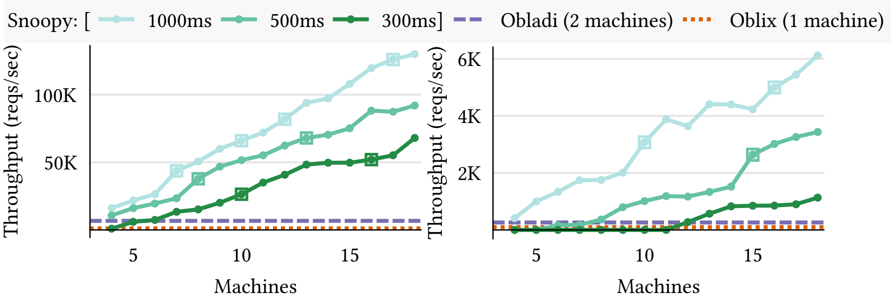
Comparison to Redis. To show the overhead of obliviousness, we also measure the throughput of Redis for 2M 160-byte objects with an increasing cluster size. For 15 machines, Redis achieves a throughput of 4.2M reqs/sec, 39.1× higher than Snoopy when configured with 1s latency. Because we pipeline Redis 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, 64] 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) [64]. 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 [66] 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 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



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

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

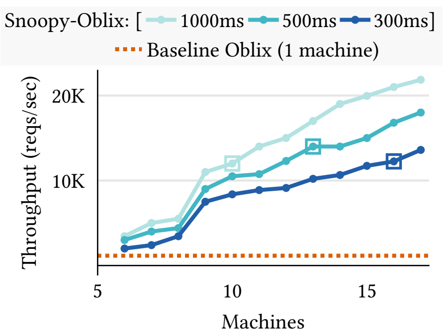
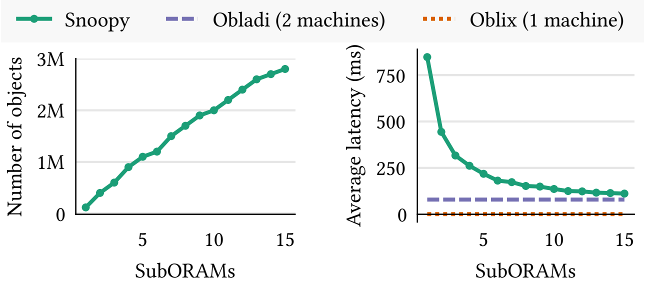


Figure 9. Snoopy achieves higher throughput with more machines. Boxed points denote when a load balancer is added instead of a subORAM. Oblix and Obladi cannot securely scale past 1 and 2 machines, respectively.

Figure 10. Throughput of Snoopy using Oblix [66] 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

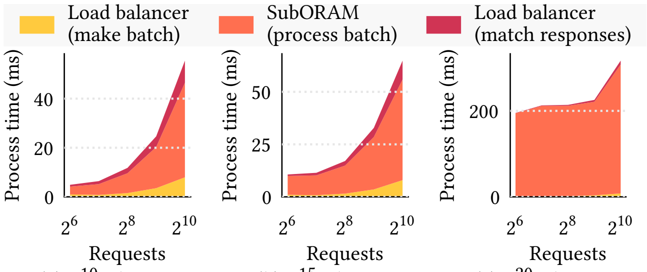
Figure 11. (a) Adding more subORAMs allows for increasing the data size while keeping the average response time under 160ms (RTT from US to Europe). (b) Adding more subORAMs reduces latency. Snoopy is running 1 load balancer and storing 2M objects.

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.

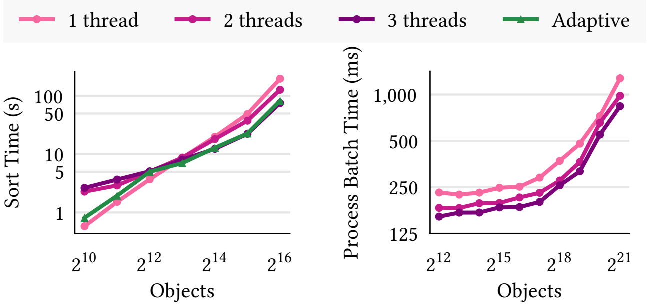


(a) 2^{10} objects

(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).

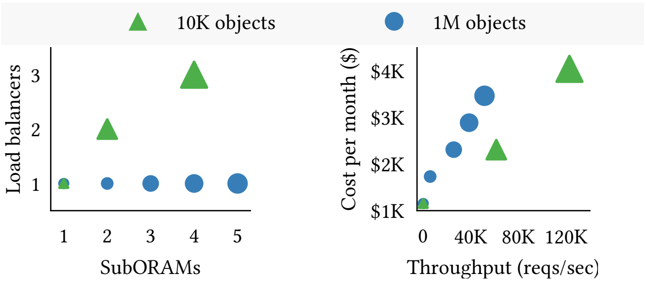


(a) Bitonic sort

(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).

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.



(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.

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 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 [74]. Prior work has explored how to defend against such attacks [11, 62]. 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 [62] 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 [47]. 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 [43, 48]. Existing systems develop relevant batching [4, 41] and preprocessing [52] 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 [66], ZeroTrace [87], Obliviate [3], Pyramid ORAM [24], and POSUP [46] do not support concurrency. Snoopy, in contrast, optimizes for throughput and leverages batching for security and scalability. ObliDB [31] supports

SQL queries by integrating PathORAM with hardware enclaves, but uses an oblivious memory pool unavailable in Intel SGX. GhostRider [59] and Tiny ORAM [33] use FPGA prototypes designed specifically for ORAM. While no general-purpose, enclave-based ORAM supports request parallelism, MOSE [45] and Shroud [60] 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 [102] 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 [86] 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 [96], 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 [80].

Scaling out ORAMs. Several ORAMs support distributing compute and/or storage across multiple servers. Oblivistore [92] distributes partitions of SSS-ORAM [93] 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 [86].

Pancake [37] 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, 67, 77, 78]. 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 [70] apply this bound in the ORAM setting). Raab and Steger [78] use the first and second moment method to derive a bound where overflow probability depends on bucket load. Ramakrishna's [81] 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. [84] 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 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, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. *IACR Cryptol. ePrint Arch.*, 2021, 2021. https://eprint.iacr.org/2021/1280.pdf.

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

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

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

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

[32] 5 advantages of a cloud-based EHR. https://www.carecloud.com/continuum/5-advantages-of-a-cloud-based-ehr-for-large-practices/.

[33] 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.

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

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

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

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

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

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

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

[46] 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.

[47] 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.

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

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

[50] 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.

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

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

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

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

[55] 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.

[56] 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.

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

[58] 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.

[59] 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.

[60] 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.

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

[62] 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.

[63] 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.

[64] 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.

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

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

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

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

[69] 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.

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

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

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

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

[74] 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.

[75] 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.

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

[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*. Springer, 1998.

[78] 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.

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

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

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

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

[83] 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.

[84] 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.

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

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

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

[88] 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.

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

[90] 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.

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

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

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

[94] 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.

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

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

[97] 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.

[98] 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.

[99] 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.

[100] 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.

[101] 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.

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

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