

Differentially Oblivious Resizing for Group-By Aggregations

James Bell-Clark, Albert Cheu, Adria Gascon, and Jonathan Katz
Google

Lukas Gerlach
CISPA Helmholtz Center for Information Security*

Abstract

Systems for private group-by aggregation (e.g., computing histograms, min/max values, or averages) let a confidential virtual machine (CVM) generate statistics from streaming user data. A particular challenge in such systems is ensuring that the CVM’s memory-access patterns do not reveal (too much) private information to the untrusted host. While it is possible to rely on oblivious RAM (ORAM), doing so imposes a significant performance penalty and requires allocating sufficient memory to handle a worst-case data stream.

We introduce ROGA, a scheme for Resizable Oblivious Group-by Aggregation. ROGA uses an extension of oblivious single-access machines and thus improves performance, both asymptotically and concretely, relative to using ORAM. It also incorporates a novel, *differentially oblivious* resizing mechanism that ensures the allocated memory is within a constant factor of the memory used by a non-oblivious solution. ROGA also parallelizes cleanly across multiple cores for improved performance.

We implement ROGA in Rust and use Binsec/Rel to verify trace noninterference of its compiled fixed-capacity operations, resize estimator, and fixed-work noise sampler, showing that executions with equal public parameters have identical branch-target and memory-address traces for all secret inputs. The only data-dependent branch is the differentially private resize decision. Compared to state-of-the-art oblivious schemes for confidential analytics (which do not support private resizing), ROGA is up to 50.4× faster when sharded across 64 logical cores, and private resizing cuts memory by up to 14× compared to domain-provisioned instances. In a case study, ROGA using 16 cores processes the standard network statistics of a 277-million-packet backbone trace with 5.3× end-to-end overhead over a non-oblivious pipeline while exactly matching the reference output.

1 Introduction

Systems for confidential analytics [13, 69, 29, 23, 35, 2, 28, 17] use trusted execution environments (TEEs) to generate statistics from private user data. We focus on confidential virtual machines (CVMs), a type of TEE that protects an entire virtual machine. In the particular case where the desired statistics can be generated via group-by aggregation, the CVM has access to

*Work done while visiting Google.

a stream of items in the form of key-value pairs $(key_1, val_1), \dots$, and wants to maintain summary statistics of the form

$$\left\{ \left(key, f(\{val_i : key_i = key\}) \right) \right\}$$

for an associative and commutative function f , e.g., the CVM may maintain a histogram of the number of items in the stream associated with each key. Assuming items in the stream are encrypted under a public key whose corresponding private key is held internally by the CVM, and the computation could be done entirely within attacker-unobservable memory inside the CVM, we can ensure differential privacy [26] of the entire computation if we have the CVM add calibrated noise to the histogram before it is released. Attestation lets users verify the software being run on the CVM that enforces these protections.

In practice, a CVM does not have sufficient memory to run the entire computation internally, and must use encrypted guest memory during its execution. Despite encryption, accesses to that memory remain observable, and it is well known that the memory-access patterns leak information. For example, if the CVM uses a conventional hash table, then an adversary observing memory accesses can detect when two items share a key. Recent work demonstrates this attack on commercial CVMs [67].

Several works have proposed using oblivious RAM (ORAM) [32] to hide memory-access patterns. EnigMap [59] (and its Obliviouslabs descendant [46]) and H2O2RAM [68] are oblivious maps designed specifically for TEEs where, unlike in the traditional client-server setting, *doubly oblivious* ORAM is needed because the attacker-unobservable memory inside the CVM is limited to CPU registers and caches. Using ORAM, however, imposes a significant performance penalty. Asymptotically optimal ORAM [7] requires $\Omega(\log N)$ physical memory accesses for each logical memory access (where N is the size of the logical memory), but its large constants make it impractical; more practical ORAM schemes [57, 51] incur $\Omega(\log^2 N)$ overhead.

Another drawback of using standard ORAM schemes is that they are typically provisioned for some fixed logical memory capacity N . In our setting, this would mean allocating memory proportional to the maximum possible number of distinct keys in the stream, which might be extremely wasteful: if only d distinct keys are actually present, then $O(d)$ storage would have been sufficient. While there are resizable ORAM schemes that allow N to dynamically grow over time [45, 9], resizing operations would themselves leak information. (In fact, this resizing signal constitutes the foundation of some attacks demonstrated in the literature [67].)

1.1 Our Contribution

We introduce ROGA, a scheme for Resizable Oblivious Group-by Aggregation that addresses the drawbacks above. Conceptually, ROGA incorporates three ideas:

- We observe that for of group-by aggregation, we can rely on an extension of oblivious *single-access* machines (OSAMs) [6, 47] rather than full-fledged ORAM. Very roughly speaking, this is because we can cast all updates as writes and only need to read data at the end. OSAM can be achieved using a *non-recursive* variant of path/ring ORAM, meaning per-access overhead is only $\Theta(\log N)$ with good concrete efficiency.

- Building on prior work [61, 15, 40, 10, 14, 50], we generate the (observable) resize schedule using a differentially private mechanism. This makes ROGA’s overall access pattern *differentially oblivious* [15], while allowing resizing to be done so that physical memory usage is within a constant factor of the required logical memory (i.e., the number of distinct keys in the stream). Our resizing mechanism uses a differentially private estimate of the number of distinct elements that requires no additional memory and incurs no additional memory-access overhead. We also introduce and prove safe a lazy resizing strategy that incrementally adds new capacity, thus avoiding bulk data migration.
- We show how to scale ROGA on multiple cores by adapting the load-balancing strategy of Snoopy [22]. Here we rely on *sharding* physical memory across multiple ROGA instances. Our experiments show that this strategy scales substantially better with the number of cores than existing oblivious maps.

We summarize the properties of ROGA as follows:

Theorem 1 (Security and efficiency, informal). *For any input stream, ROGA uses memory linear in the number of distinct keys N and has¹ $\tilde{O}(\log N)$ amortized memory-access cost per update. Its final readout has cost $\tilde{O}(N)$. The observable execution is $O(1)$ -differentially oblivious.*

We implement ROGA in Rust and use Binsec/Rel [21] to verify trace noninterference of ROGA’s operations except for resizing. Specifically, for each of 23 components of ROGA, the verification proves that executions with equal public parameters have identical branch-target and memory-address traces. Resizing decisions themselves are differentially private. Our experiments show that ROGA outperforms state-of-the-art oblivious maps for block capacities from 2^{16} to 2^{26} . At the DRAM-resident block capacity of 2^{23} used for our multicore evaluation, ROGA is 8.3–9.4× faster on a single core and 27.8–50.4× faster when sharded across 64 cores. Private resizing reduces the memory footprint by up to 14× compared to an instance provisioned for the experiment’s full 4M-key domain.

A case study on network monitoring demonstrates ROGA’s practicality. With 16 cores, ROGA processes standard per-flow network statistics, packet and byte counts, and flow durations for all 277 million packets of a 15-minute backbone trace with 5.3× end-to-end overhead over a non-oblivious pipeline for the same task.

Outline. Section 2 formalizes the group-by aggregation problem, threat model, and security goals. Section 3 reviews the oblivious memory primitives used by ROGA. Section 4 presents the fixed-capacity ROGA construction; Section 5 introduces differentially oblivious resizing; and Section 6 extends ROGA with sharding for multicore scalability. Section 7 describes our implementation of ROGA and its binary-level verification. Section 8 evaluates ROGA against state-of-the-art oblivious maps, and Section 9 applies it to private network monitoring. Section 10 reviews related work.

¹We write $\tilde{O}(k)$ for $O(k \cdot \text{polylog } k)$.

2 Problem Setting and Security Goals

We define the aggregation task and its CVM deployment, model the adversary and its view of the execution, recall the privacy definitions we build on, and state our goals.

2.1 Group-By Aggregation in a CVM

Our goal is to maintain a group-by aggregation of keyed streaming data.

Definition 2 (Keyed data stream). Let $\mathcal{K}$ be a key universe and $\mathcal{V}$ a value domain, both sets of fixed-length bit strings whose widths are chosen when the data structure is initialized. An input is a stream $S = ((\text{key}_1, \text{val}_1), \dots, (\text{key}_n, \text{val}_n))$ of key-value pairs with $\text{key}_i \in \mathcal{K}$ and $\text{val}_i \in \mathcal{V}$. We write $\text{keys}(S) \subseteq \mathcal{K}$ for the set of keys appearing in S .

Definition 3 (Group-by aggregation). Let $f : \mathcal{V} \times \mathcal{V} \rightarrow \mathcal{V}$ be an associative and commutative aggregate operator. The group-by aggregation of S under f is

$$\left\{ \left(x, f(\text{val}_i : \text{key}_i = x) \right) : x \in \text{keys}(S) \right\},$$

where f applied to several values is well defined by associativity and commutativity.

The data structure supports two types of operations: an update takes a key-value pair (x, v) , appends it to the stream, and merges it into the aggregation, and a bulk read reports one aggregate for every observed key. For simplicity we assume a single read after all items have been processed. However as long as reads happen on a fixed schedule, i.e., are independent of the input data, they do not need to be destructive. Items carry values in $\mathcal{V}$, and any internal aggregate state of the data structure, such as counters, min/max values, or sample moments, is an implementation detail. The key universe is fixed at initialization, but the number of its elements that appear in a particular stream is not known in advance. Rather than provisioning physical capacity for all of $\mathcal{K}$, ROGA grows it to match the distinct keys actually observed.

We intend to perform the group-by aggregation in a confidential virtual machine. It runs a guest whose memory is to the host only visible in encrypted form, as realized by AMD SEV-SNP and Intel TDX [3, 4, 34]. A CVM can use remote attestation which proves a measurement of the software running in the guest to a remote verifier [4]. Remote attestation allows the verifier to check that the CVM runs the code the verifier expects. Deployed systems already run analytics over user data without exposing the raw records to the operator [28, 17].

2.2 Threat Model

Our adversary is a malicious CVM operator. While memory encryption protects the CVM guest against the hypervisor reading its data, the hypervisor can still access side-channel information about the guest. A wide range of side channels have been established on CVMs, among the most persistent and mature ones being single-stepping-based ones and page fault attacks [60, 63, 66]. These attacks allow precise monitoring of which code the victim executes and which data pages it touches. Such attacks have also been applied to confidential data analysis,

where it was shown that the side-channel information an attacker gains is enough to break an algorithm’s DP guarantees inside the CVM [67, 11].

We explicitly exclude power and other side channels that memory obliviousness cannot mitigate. Mitigating such side channels requires different techniques that are not feasible for arbitrary code [38]. Similarly, fault attacks that alter ROGA’s internal operations to force leakage are out of scope.

We also exclude a physical adversary who interposes on the memory bus. Such an adversary reads the address-tweaked ciphertext directly off the DRAM bus and recovers data regardless of any access-pattern defense [19], but this requires physical access to the machine and lies outside the remote CVM-operator model we target.

We target processors on which the ciphertext side channel is closed for this remote adversary, either because the hardware withholds host read access to guest ciphertext [5] or because guest-private pages are not host-readable at all. The ciphertext side channel would otherwise let the host recognize that the same plaintext was written back to the same physical address [37, 36]. This channel is orthogonal to the access pattern that ORAM hides; it affects every CVM workload rather than oblivious ones in particular, and it is the subject of its own line of hardware and software mitigations [36, 24].

The adversary knows the public parameters: the stream length $n = |S|$, the tree’s structural parameters (bucket size, eviction interval, and initial height), and the privacy parameters. The stream length is public in any case, since the eviction schedule makes the number of processed updates observable. The adversary observes $\text{Accesses}(M, S)$. Memory encryption hides the contents of guest memory, so the adversary sees which addresses are touched but not what they hold. Stream contents reach the CVM under public-key encryption and are decrypted only inside it.

Page-fault tracking, cache attacks, and single stepping [66, 63, 62] expose $\text{Accesses}(M, S)$ at instruction granularity. Because the trace includes instruction fetches, it captures control flow: a secret-dependent branch fetches different addresses. We formalize the adversary’s complete view below.

Definition 4 (Access pattern). For a randomized implementation M processing stream S , the *access pattern* $\text{Accesses}(M, S)$ is the ordered sequence of physical addresses touched during execution, including instruction fetches and data accesses labeled as reads or writes.

2.3 Differential Privacy & Obliviousness

Differential privacy (DP) bounds how much any one individual’s data can change the distribution of a randomized computation’s output [26].

Definition 5 (Neighboring streams). Two streams $S = ((\text{key}_i, \text{val}_i))_{i=1}^n$ and $S' = ((\text{key}'_i, \text{val}'_i))_{i=1}^n$ are *neighboring* if there is exactly one index i with $(\text{key}_i, \text{val}_i) \neq (\text{key}'_i, \text{val}'_i)$.

A randomized mechanism M is (ϵ, δ) -differentially private if, for all neighboring streams S, S' and output events E ,

$$\mathbb{P}[M(S) \in E] \leq e^\epsilon \mathbb{P}[M(S') \in E] + \delta.$$

Here ϵ bounds how much more likely any observation can become when one contribution changes, and δ is a small probability with which that bound may fail. Under basic composition, both parameters are additive across releases, and DP is closed under post-processing [27]. Finally, the above implies a per-individual guarantee if each individual contributes one key-value pair.

The adversary handled by the standard differential privacy definition exploits only the output of M . In our work, we instead shift focus to the adversary that monitors its access pattern. Our defense against this adversary is to make the access pattern differentially private as a function of the stream, a property known as *differential obliviousness* [15].

Definition 6 (Differential obliviousness). An implementation M is (ϵ, δ) -*differentially oblivious* if for every pair of neighboring streams S, S' and every set E of access patterns,

$$\mathbb{P}[\text{Accesses}(M, S) \in E] \leq e^\epsilon \cdot \mathbb{P}[\text{Accesses}(M, S') \in E] + \delta.$$

Full obliviousness is the special case $\epsilon = \delta = 0$.

Output privacy requires a separate mechanism applied to ROGA’s exact aggregate; it composes with differential obliviousness (Definition 6) but lies outside our scope.

2.4 Goals

Our security goal is to make ROGA’s execution differentially oblivious. Except for resizing, ROGA follows a fixed schedule of memory accesses that depends only on public parameters and the stream length, not on the stream’s keys or values. Resizing changes the allocated memory and is observable to the host, so the only input-dependent part of the access pattern is when resizes occur. ROGA chooses these resize times using a differentially private mechanism. Consequently, it achieves differential obliviousness.

This guarantee protects the execution of the aggregation, but not the statistics it releases. Making that aggregate itself differentially private, by bounding per-user contribution and adding calibrated noise to the reported values, is the responsibility of the caller and is handled by existing differentially private query systems [41, 64]. Beyond this security guarantee, ROGA has two further goals: correctness and efficiency.

Correctness requires that ROGA report the true aggregate for every stream over the fixed key universe, even if every item has a different key or every key in the universe appears. The structure’s initial capacity may nevertheless be smaller than $|\mathcal{K}|$; it doubles its capacity whenever the load estimator fires, and every update is merged into the aggregate for its key, so a completed run reports the true aggregate for every observed key. The one failure mode is a stash that outgrows its fixed capacity, which aborts the run rather than returning a wrong answer, and which happens with negligible probability. This is a correctness and availability failure: accesses remain oblivious until the abort, and an observable abort contributes only its failure probability to the δ term in Definition 6.

Efficiency has two parts. Resizing should keep ROGA’s memory use proportional to the realized number of distinct keys, rather than to the size of the fixed key universe or the stream

length n . The total number of memory accesses, equivalently the amortized accesses per ingested update, should stay close to a fixed-capacity structure sized for that realized key count. Merging serves both, since it keeps one block per distinct key rather than one per item.

3 Background: Oblivious Memory

This section reviews the oblivious memory primitives on which ROGA builds and explains why none directly supports our group-by aggregation workload.

(Doubly) Oblivious RAM. Oblivious RAM (ORAM) [32] makes the physical access pattern independent of the logical addresses, so an observer learns only the number of accesses, not the accessed locations. ORAM supports arbitrary sequences of reads and writes to arbitrary addresses. Traditional ORAM schemes assume a client-server setting: a client dispatches read and write requests to an honest-but-curious server, which faithfully executes them but may observe read and write patterns. The adversary does not see what the client does to data on its end, which means client-side memory access patterns can be data-dependent. In contrast, our adversary also observes how the client-side ORAM code accesses its own auxiliary data structures. Prior work has called an algorithm that ensures obliviousness against our adversary *doubly oblivious* [44], because one could simulate it by ensuring client-side memory access patterns are data-independent while also making server-side ones data-independent. Our threat model (Section 2.2) has no private client memory, so any scheme in our setting must be doubly oblivious. This changes what is expensive: the implementation performs more linear scans with constant-time conditional moves rather than indexed lookups, and every subroutine that touches secret data must be branch- and address-oblivious. Doubly oblivious maps are the standard design for enclaves [53, 44, 59, 68].

Tree-based ORAM. Path ORAM [57] stores encrypted blocks in Z-slot buckets of a binary tree with N leaves, one per logical location. Blocks map to random leaves and may occupy any bucket on their root-to-leaf paths. Unplaced blocks stay in a small client stash; a position map records each block’s leaf. On access, the client reads the path into the stash, assigns the block a fresh leaf, and writes it back, placing stashed blocks as deeply as possible. Fresh leaves unlink repeated accesses. Recursively storing the linear-size map adds a lookup per level, raising overhead to $O(\log^2 N)$.

Path ORAM reads and rewrites a complete path on every logical access. Ring ORAM [51] reduces this online bandwidth by reading only one block from each bucket on the path and deferring the full-path write-back to a separate eviction. It performs one eviction every A accesses along paths in the deterministic reverse-lexicographic order of Gentry et al. [30], so the eviction path depends only on a public counter. Increasing A amortizes each eviction over more accesses but leaves more blocks waiting in the stash.

Resizable ORAM. Resizable ORAM allows the physical tree to grow or shrink while preserving ORAM security [45, 9]. Prior constructions expose the client-issued resize request and therefore the time at which a resize occurs. This is sufficient when capacity follows a public schedule, but resizing based on the number of distinct keys would reveal information about the stream. A scheme that resizes based on private data must therefore hide not only the accesses performed during a resize, but also the decision of when to resize.

Oblivious single-access machines. Path ORAM remaps a block on every access because the block may be read again, and the fresh leaf keeps repeated accesses unlinkable. A workload that reads each address at most once can skip remapping, since there is no second access to unlink. Oblivious single-access machines (OSAMs) support exactly this model: each address is written at most once and read at most once [6]. On allocation, the OSAM samples a leaf and returns a handle encoding it. The caller supplies this handle at read time; with no later access to unlink, the block needs no reassignment. Thus OSAM needs no position map eliminating the main source of Path ORAM’s $O(\log^2 N)$ overhead.

Multi-write OSAMs allow repeated writes to an address before its single read, which returns the latest value [47]. Several versions of one address may temporarily coexist in the tree and stash. The scheme identifies the latest version and discards stale copies using private client memory while retaining the same fixed-leaf allocation.

Gap for group-by aggregation. The group-by workload of Section 2 resembles a multi-write, single-read workload, but it does not fit the interface of existing multi-write OSAMs. Those schemes retain only the latest value written to an address, whereas group-by aggregation must combine all values associated with the same key. Implementing an update through the existing interface would require reading the current aggregate before writing the new value, which violates the single-read restriction.

4 Fixed-Size ROGA

Our algorithm consists of three main components. The first is a keyed, aggregation-aware adaptation of multi-write OSAM (Section 3), which merges group-by updates inside the tree. The second is a differentially private estimator that determines when to resize, along with an efficient resize procedure. The third shards ROGA into independent trees to scale it across many cores. This section presents the single-tree construction at a fixed capacity, Section 5 presents the resizing mechanism, and Section 6 collects everything related to sharding.

Unlike multi-write OSAM [47], which removes overwritten copies in private client memory, ROGA combines repeated keys inside the tree without such memory. One order handles merging and placement, avoiding a second oblivious sort.

A group-by aggregate data structure holds one entry per distinct key, so it should grow with the number of distinct keys rather than with the stream length n . ROGA achieves this by keeping only one block per distinct key and by growing the tree as more space is required (Section 5). This section describes the construction at a fixed capacity. Algorithm 1 gives the control flow for a single tree. It includes the resizing hook, the epoch and sweep bookkeeping, and the calls to `ShouldResize` and `StartResize` explained in Section 5.

The tree starts empty, and its buckets are padded to Z slots with dummy blocks $\perp$. The stash is a multiset of blocks (x, v, e) , and $\cup$ denotes multiset union. Algorithm 1 processes the stream in order with public update index i . The sweep eviction interval $\tilde{A}$ equals A unless stated otherwise.

Layout and routing. ROGA keeps the binary tree layout of Path ORAM [57]. The tree has height L (so 2^L leaves), and its nodes are buckets holding Z blocks each; its epoch target is $N = D2^L$ distinct keys, with $D = 25$ in our resizing experiments. It additionally has a small

```

def Init( $L_0$ ):
     $L \leftarrow L_0$ 
     $\text{evictCtr}, \text{epoch}, \text{sweepLeft} \leftarrow 0$ 
     $c \leftarrow []$ 
     $\text{stash} \leftarrow \emptyset$ 
     $k \leftarrow \text{BeginEpoch}(L, D2^L)$  Algorithm 3
def Update( $\text{key}_i, \text{val}_i$ ):
     $\text{stash} \leftarrow \text{stash} \cup \{(\text{key}_i, \text{val}_i, \text{epoch})\}$ 
     $A' \leftarrow \tilde{A}$  if  $\text{sweepLeft} > 0$ , and  $A$  otherwise
    if  $i \bmod A' = 0$  then
        | Evict()
def StartResize():
     $L \leftarrow L + 1$  double tree capacity
     $\text{epoch} \leftarrow \text{epoch} + 1$ 
     $\text{sweepLeft} \leftarrow 2^{L-1}$ 
     $k \leftarrow \text{BeginEpoch}(L, D2^L)$  Algorithm 3

def Evict():
     $\ell \leftarrow \text{evictCtr} \bmod 2^L$ 
     $\text{evictCtr} \leftarrow \text{evictCtr} + 1$ 
     $c' \leftarrow \text{EvictPath}(\ell, \text{sweepLeft} > 0)$  Algo. 2
    append  $c'$  to list  $c$ 
    if  $\text{sweepLeft} > 0$  then
        |  $\text{sweepLeft} \leftarrow \text{sweepLeft} - 1$ 
    if  $|c| = k$  then
        | if ShouldResize( $c$ ) then StartResize()
        |  $c \leftarrow []$ 
def Readout():
     $Y \leftarrow$  all blocks in stash and the tree
    sort  $Y$  by  $(\mathbb{1}[x_j = \perp], x_j)$ 
    for  $j \leftarrow 1$  to  $|Y| - 1$  do
        | if  $x_{j-1} = x_j \neq \perp$  then
            |  $(v_j, x_{j-1}) \leftarrow (f(v_{j-1}, v_j), \perp)$ 
    Compact( $Y$ ) real blocks, then dummies [54]
    return  $Y$  fixed-length, encrypted, and padded

```

Algorithm 1: Single-tree ROGA control flow. Buffered updates are periodically evicted; read-out returns a fixed-length padded aggregate; a private load test may start lazy resizing. Intervals are A normally and $\tilde{A}$ during sweeps.

stash for blocks that do not fit in the tree. As stated in Section 2.2, there is no trusted client memory, so all operations on the stash are performed with oblivious algorithms. A block is a single aggregation entry: a key, a value (the aggregate state), and an epoch counter, which is required for the resizing operations in Section 5.1. A block with key $\perp$ is a dummy. Ingesting a pair (x, v) labels it with $F_K(x)$, where F is a pseudorandom function and K a secret key internal to the CVM. The least significant L bits of the label index the leaf the block routes to. Updates enter the stash, and evictions perform all data movement. Unlike Path ORAM, which remaps every access and requires a recursive position map, ROGA derives each leaf from the key, adapting OSAM’s map-free allocation [6, 47].

Eviction and merge. Ordinarily, ingestion buffers writes for A updates before evicting one path, following Ring ORAM [51]. An eviction reads the path from the root to a scheduled leaf. Paths follow the reverse-lexicographic order of Gentry et al. [30]. Because labels are read from the low bits up, $\text{evictCtr} \bmod 2^L$ enumerates that order directly. The counter is not reset during resizing; increasing L continues the same schedule over the enlarged tree. The eviction set and the stash form a working set of $W = (L + 1 + s)Z + R$ blocks, where $s = 1$ during a resize sweep and $s = 0$ otherwise, and R is the fixed per-tree stash capacity, a few hundred blocks in total for the parameters of Section 8. The eviction has two jobs on this working set. It merges blocks that share a key into one block, and it pushes every surviving block as deep into the eviction set as possible. A block’s eligible buckets are those in the eviction set that lie on its labeled path, and occupancy may force it into a shallower one. The merge is what

```

def EvictPath( $\ell$ , sweepActive):
     $E \leftarrow$  buckets on the root-to- $\ell$  path
    if sweepActive then  $E \leftarrow E \cup \{\text{Sibling}(\ell)\}$ 
    for  $u \in E$  do
        |  $\text{stash} \leftarrow \text{stash} \cup \text{ReadBucket}(u)$ 
     $c \leftarrow \text{SortAndMerge}(\text{stash}, \ell)$ 
    PlaceSet( $E$ )
    return  $c$ 

def PlaceSet( $E$ ):
     $n_u \leftarrow 0$  for each  $u \in E$       per-bucket occupancy
    for  $i$  with  $x_i \neq \perp$  do      sorted order, deepest first
        |  $\text{leaf}_i \leftarrow \text{LeafBits}(F_K(x_i), L)$ 
        |  $t_i \leftarrow$  deepest non-full  $u \in E \cap \text{Path}(\text{leaf}_i)$ 
        | if  $t_i$  exists then  $n_{t_i} \leftarrow n_{t_i} + 1$ 
    Compact(stash)      real blocks and destinations to front [54]
    DistributeToSet(stash,  $E; (t_i)_i$ ) [54]

def SortAndMerge(stash,  $\ell$ ):
    for  $i \leftarrow 0$  to  $|\text{stash}| - 1$  do
        |  $b_i \leftarrow F_K(x_i)$ 
        |  $\text{leaf}_i \leftarrow \text{LeafBits}(b_i, L)$ 
        |  $\text{order}_i \leftarrow \text{OrderBits}(b_i, L)$ 
        |  $k_i \leftarrow (\mathbb{1}[x_i = \perp], \text{rev}(\text{leaf}_i \oplus \ell), \text{order}_i)$ 
    sort stash by  $k_i$ 
    for  $i \leftarrow 1$  to  $|\text{stash}| - 1$  do
        | if  $x_{i-1} = x_i \neq \perp$  then
            |  $v_i \leftarrow f(v_{i-1}, v_i)$ 
            |  $e_i \leftarrow \min(e_{i-1}, e_i)$ 
            |  $x_{i-1} \leftarrow \perp$ 
    return number of  $(x, v, e) \in \text{stash}$  where
        Leaf( $F_K(x)$ ) =  $\ell \wedge e = \text{epoch}$ 

```

Algorithm 2: Single-sort path eviction. `EvictPath` reads the scheduled eviction set, merges duplicate keys, and places surviving blocks as deeply as possible using one oblivious sort. It returns the current-epoch count for leaf ℓ to the load estimator; unplaced blocks remain in the stash. During a lazy sweep, the new sibling has its own capacity counter and is written back with the path, as required by Lemma 12.

separates ROGA from a generic ORAM: duplicates are combined with the aggregate operator f of Definition 3, so the tree holds one block per distinct key rather than one per item. Merging requires equal keys to be adjacent, while placement requires blocks to be ordered by eligible depth. A naive construction would satisfy these requirements with two oblivious sorts, each requiring $\mathcal{O}(W \log^2 W)$ conditional swaps [42, 59].

Single-sort eviction. ROGA avoids the second sort by encoding both requirements in one composite sort key. For a tag $b = F_K(x)$ and eviction leaf ℓ , let $\text{LeafBits}(b, L)$ contain its routing bits and $\text{OrderBits}(b, L)$ its remaining ordering bits (Section 7.1). We sort each block by

$$(\mathbb{1}[x = \perp], \text{rev}(\text{LeafBits}(b, L) \oplus \ell), \text{OrderBits}(b, L)).$$

The reversed XOR orders blocks by decreasing eligible depth, while the remaining bits make copies of the same key contiguous. The same order also handles the sibling bucket included during a resize sweep.

Algorithm 2 then merges each equal-key run with f , retaining the key’s first-entry epoch, and greedily assigns every survivor to its deepest available bucket. Oblivious compaction and distribution write the result back in $\mathcal{O}(W \log W)$ work [54]. Compared with the avoided $\mathcal{O}(W \log^2 W)$ sort, this saves a factor $\log W$ in that stage, about 10–11 for our evaluated capacities. Thus, each eviction uses one $\mathcal{O}(W \log^2 W)$ sorting network instead of the two required by a naive construction, halving its sorting-network work. All subsequent operations use fixed scans or oblivious networks, so their access patterns are independent of the data.

Readout. Reads are deferred to a readout that runs on a fixed public schedule, once after the

stream ends or at regular points during it. A readout scans every bucket and the stash in a fixed order, sorts the collected blocks by key, merges each equal-key run with f , and obviously compacts the surviving blocks to the front. It emits the entire fixed-length array, including dummy padding, in encrypted form; the trusted recipient discards the dummies after decryption. Thus, the readout reveals no additional information about the number of distinct keys beyond the public capacity determined by the differentially private resize schedule. Point queries for individual keys are not supported, since the deterministic labels would reveal repeated accesses to the same key.

5 Differentially Oblivious Resizing

ROGA doubles its tree height when the occupancy estimate approaches capacity. Resizing requires allocating new physical memory, an event the CVM host can inherently observe. If we resized exactly when the distinct-key count crossed a capacity boundary, the timing of the allocation would leak the precise point in time when that count was reached. Thus, the allocation schedule must be differentially private. Section 5.1 describes how the tree grows and how the load estimator obtains its samples. Section 5.2 bounds the error of the estimator, and Section 5.3 gives the private threshold test together with the two guarantees: the schedule of resize times is DP, and with high probability a resize fires before the tree overflows, and only once the tree is within $O(N^{2/3})$ of its capacity target. Finally, Section 5.4 shows how the added capacity becomes usable without extra evictions.

5.1 Growing the Tree and Estimating Load

Resizing poses two challenges: how to grow without disturbing the existing blocks, and how to decide when to grow.

Doubling without data movement. Level j routes on bit j of the label, reading from the low bits up. Doubling the tree adds one level, which consumes one more bit. Every block’s old path is therefore a prefix of its new one, so the block still lies on the path to its leaf. The tree therefore doubles without moving blocks, enabling the lazy growth of Section 5.4.

Deciding when to grow. It is preferable to grow when the distinct keys the tree currently holds are close to its capacity. We therefore introduce a load estimator that tracks that count using information the evictions already expose. For efficiency, we embed such an estimator within our tree data structure, particularly in the leaves. Note that each non-dummy block has one destination leaf, fixed by its label, so the leaves partition the blocks into disjoint groups. At any point in time, a leaf’s *count* is the number of blocks that are intended for that leaf, i.e., blocks in the tree whose label matches that leaf. We call the interval between resizes an *epoch*. As Section 4 established, each block records the epoch when its key first entered the tree. The estimator $\hat{d}$ counts only blocks from the current epoch, as required by the privacy argument in Section 5.3.

Assuming we are in epoch e , let $d_{e,t}$ denote the number of distinct keys that first entered the tree in epoch e by update t . Because the PRF routes keys uniformly, the count of a single leaf is distributed as $\text{Bin}(d_{e,t}, 2^{-L})$, where $\text{Bin}(n, p)$ denotes the binomial distribution over n trials

def BeginEpoch(L, N): $k \leftarrow \lfloor (N2^L/A^2)^{1/3} \rfloor$ $\Delta \leftarrow 2^L/k$ $\beta(\alpha) \leftarrow \sqrt{3N\Delta \ln(1/\alpha)}$ $\tau \leftarrow N - (1+r)Ak - \beta(\alpha)$ $\zeta(x) \leftarrow (1-x)\text{-quantile of Lap}(6\Delta/\varepsilon)$ $\theta \leftarrow \text{Lap}(6\Delta/\varepsilon)$ $\hat{\tau} \leftarrow \tau + \theta - \zeta(\alpha) - \zeta((2\alpha)^r)$ return k	def ShouldResize($c_1, \dots, c_k$): $\hat{d} \leftarrow \Delta \sum_{i=1}^k c_i$ $\eta \leftarrow \text{Lap}(6\Delta/\varepsilon)$ return $(N/2 + \hat{d} + \eta > \hat{\tau})$ <i>Boolean value</i>
---	--

Algorithm 3: Differentially private load estimator. Its parameters are privacy budget ε , ordinary eviction interval A , resize failure probability α , and number r of guaranteed resize attempts. BeginEpoch runs at the start of every epoch, on the public tree height L and capacity target N ; ShouldResize runs once per measurement interval, on the k eviction counts of that interval.

with success probability p . An eviction to leaf ℓ already touches the path and the stash, which together hold every block destined for ℓ . It therefore observes ℓ 's count c for free. A count is exact for its own leaf, but extrapolating from one leaf to the whole tree is noisy. The estimator therefore pools k consecutive evictions, a *measurement interval*, and rescales their counts to $\hat{d} = (2^L/k) \sum_{i=1}^k c_i$. Since $k < 2^L$, the bit-reversal schedule guarantees that the k evictions visit distinct leaves, so the accumulated count is binomially distributed. Operationally, the estimator adds a running sum and interval counter and invokes the private comparison of Section 5.3 once per measurement interval. Section 5.2 derives k by balancing the error terms.

The randomized growth test prevents a resize from revealing exactly when the count crossed the threshold. Because earlier-epoch keys are invisible to the estimator, the comparison adds $N/2$ to $\hat{d}$, treating the previous tree as full. Algorithm 1 keeps the bookkeeping, collecting each eviction's count and doubling the height when the test fires. Algorithm 3 gives the private estimator; Section 5.3 proves its schedule is DP and prevents tree overflow.

5.2 Estimator Error

Throughout this section, we focus on one epoch e of the algorithm. Let L denote the tree height, $N = D2^L$ the epoch's distinct-key capacity target, A the maximum eviction interval, and recall $d_{e,t}$ is the number of novel keys input in the epoch up to t . Two error sources separate $\hat{d}$ from the current load. First, the estimate is stale. The count for leaf ℓ_i was taken when that leaf was evicted, so a key that later routes to ℓ_i is missing from it. The oldest count of an interval is $A(k-1)$ updates old, and at most one fresh key arrives per update, so $\hat{d}$ can lag behind $d_{e,t}$ by up to Ak . Second, the estimate is a sample, with a binomial deviation of $\Theta(\sqrt{N2^L/k})$.

The two move in opposite directions in k . A longer interval means older counts, so staleness grows like Ak . It also means more leaves sampled, so the deviation falls like $\sqrt{N2^L/k}$. The larger of the two is smallest where they cross,

$$Ak = \sqrt{N2^L/k}, \quad \text{giving} \quad k = (N2^L/A^2)^{1/3},$$

and Algorithm 3 takes $k = \lfloor (N2^L/A^2)^{1/3} \rfloor$. For $2^L = \Theta(N)$ both terms are $\Theta(N^{2/3})$, the headroom of Corollary 11.

5.3 Threshold Test

Algorithm 3 splits the test in two. `BeginEpoch` runs at the start of each epoch, on the public L and N . It returns the measurement interval k to Algorithm 1; the remaining quantities stay internal to the estimator. It computes the deterministic threshold τ , draws a single threshold noise θ , and sets the internal noisy threshold $\hat{\tau}$. `ShouldResize` runs once per measurement interval, on the k eviction counts of that interval. It forms $\hat{d}$, draws a fresh noise η , and reports whether $N/2 + \hat{d} + \eta$ exceeds $\hat{\tau}$. An implementation never holds the k counts, since a running sum and a counter suffice. Both noise draws use $\text{Lap}(3 \cdot 2^{L+1}/(k\varepsilon))$.

The deterministic part of the threshold, $\tau = N - (1+r)Ak - \beta(\alpha)$ subtracts a staleness margin of Ak , a number of attempts margin rAk and the Chernoff deviation β of the sampling error at failure probability α . The staleness margin is because within an interval $\hat{d}$ lags $d_{e,t}$ by at most Ak , as Section 5.2 shows. Each negative answer delays the next decision by Ak updates, during which at most Ak fresh keys can arrive; the rAk margin therefore guarantees at least r calls to `ShouldResize`. The Chernoff term bounds how far $\hat{d}$ falls below its own mean, and it fails with probability $\leq \alpha$.

The final threshold is $\hat{\tau} = \tau + \theta - \zeta(\alpha) - \zeta(\alpha^r)$, where $\zeta(x)$ is the $(1-x)$ -quantile of the noise distribution.

Privacy. To reiterate, differential obliviousness of ROGA follows if the algorithm determining the resize schedule is differentially private, because ROGA is otherwise oblivious. We formalize our privacy claim below.

Theorem 7. *The schedule of resize times produced by Algorithm 3 is $3\varepsilon/2$ -DP with respect to neighboring streams (Definition 5).²*

We defer the full proof to Appendix D, but sketch the main ideas here. Given the index i where two neighboring streams S, S' differ, we categorize every resize based on its timing relative to i . Resizes that occur before i are based on identical data and therefore incur no privacy cost. We then argue that the first resize after i is $\varepsilon/2$ -DP by adapting the AboveThreshold mechanism [27, 11]: although `ShouldResize` is invoked many times, each noising $\hat{d}$ with fresh Laplace noise, we use *correlated* noise θ to argue that the privacy budget is spent only when the function returns ‘true.’

What about the budget consumption of subsequent resizes? Suppose S has a at index i while S' has b instead. We argue that only two other resizes could consume privacy budget: the one that ends the epoch containing the next instance of a and the one that ends the epoch containing the next instance of b . This is because every `ShouldResize` computation is affected by the *novelty* of input keys. If an instance of a or b is not the first instance, regardless of whether S or S' was input, then switching inputs does not change the $\hat{d}$ that `ShouldResize` computes. In turn, it would not change the distribution of the resize timing. Conversely, if an instance of

²We acknowledge the mismatch between ε and the proved privacy parameter; a future revision of our work will adjust our pseudocode and implementation to target ε -DP.

a or b could be the first instance, then the distribution does change; there are only at most two such cases, so that there are a maximum of three expenditures of $\varepsilon/2$.

Correctness. We show that with high probability we always resize before the number of items in the tree exceeds its capacity. Both this guarantee and the memory-efficiency guarantee below reduce to one accuracy bound on the estimator.

Lemma 8 (Estimator accuracy). *Let $\beta(\alpha) := \sqrt{3N\Delta \ln(1/\alpha)}$. For an interval ending at t ,*

$$d_{e,t} - Ak - \beta(\alpha) \leq \hat{d} \leq d_{e,t} + \beta(\alpha),$$

where each inequality fails with probability at most α .

Proof sketch. For an eviction at time s , let $c_{[t'] \rightarrow s}$ denote the number of distinct items among the first t' updates that map to the leaf evicted at time s . Counts only grow with the first index, so each count accumulated for the estimate is sandwiched as $c_{[t-Ak] \rightarrow s} \leq c_s = c_{[s] \rightarrow s} \leq c_{[t] \rightarrow s}$. For fixed t' , the sum of the bounds $c_{[t'] \rightarrow s}$ over the interval is distributed as $\text{Bin}(d_{e,t'}, k2^{-L})$, because the k evictions visit distinct leaves. A Chernoff bound and rescaling by $2^L/k$ keep the rescaled sum within $\sqrt{3d_{e,t'}\Delta \ln(1/\alpha)} \leq \beta(\alpha)$ of $d_{e,t'}$ in each direction, each side failing with probability at most α . The endpoint $t' = t$ gives the upper bound. The endpoint $t' = t - Ak$ gives $\hat{d} \geq d_{e,t-Ak} - \beta(\alpha)$, and at most Ak fresh items arrive between times $t - Ak$ and t , so $d_{e,t-Ak} \geq d_{e,t} - Ak$.

We will use the above to prove that a resize is unlikely to happen too late (Lemma 9) or too early (Lemma 10). Full proofs can be found in Appendix D.

Lemma 9. *For any epoch, if $r \leq 2^L/k$, resizing occurs before $d_{e,t} > N/2$, except with probability $2(2\alpha)^r$.*

Proof sketch. Once $d_{e,t}$ reaches $N/2 - rAk$, at least r calls to `ShouldResize` occur before it exceeds $N/2$. At each call, the estimator and measurement-noise tails fail with probability at most 2α . The calls use disjoint leaf sets, whose dependence can only reduce the probability that every call misses, giving $(2\alpha)^r$; the threshold-noise tail contributes another $(2\alpha)^r$.

Memory efficiency. We show that with high probability resizes don't occur too often. For a prefix of T updates containing M distinct keys, Corollary 11 bounds the allocated logical capacity by $2M + O(M^{2/3})$, suppressing logarithmic factors.

Lemma 10. *If $d_{e,T} \leq N/2 - (1+r)Ak - \beta(\alpha) - \zeta(\alpha) - \zeta((2\alpha)^r) - \zeta(\xi) - \beta(\xi/\lceil T/Ak \rceil) - \zeta(\xi/\lceil T/Ak \rceil)$, then the probability of a resize occurring before time T is $\leq 3\xi$.*

Proof sketch. There are at most $\lceil T/Ak \rceil$ checks by time T . A union bound controls the upper tails of every estimator and measurement-noise draw and the lower tail of the epoch's threshold noise, with total failure probability 3ξ . Substitution into the resize test gives the condition in the statement.

Corollary 11 (Capacity headroom). *Let $M = d_T$ be the number of unique keys among the first T updates. Except with probability 3ξ , the tree will not resize once it has capacity $M + O(M^{2/3} + M^{1/2} \log(T/(\xi N^{2/3})) + \log(1/(2\alpha)^r))$. Suppressing logarithmic factors, this in turn implies the capacity will not exceed $2M + O(M^{2/3})$, except with small probability.*

5.4 Lazy Resizing

Section 5.1 established that adding a level requires no data movement. Making the new capacity usable immediately would, however, require 2^L extra evictions to push all the existing blocks into the new deepest level. Algorithm 1 instead performs a *sweep*: each scheduled eviction activates the new bucket on its path, and its sibling, and the bit-reversal schedule activates the entire level after 2^L evictions. During a sweep, evictions occur every $\tilde{A} \leq A$ updates, completing it within $\tilde{A}2^L$ insertions. Our configuration sets $\tilde{A} = 10$ instead of $A = 20$, doubling the eviction rate during sweeps to limit fresh-key arrivals before full capacity. The resize times determine the sweep schedule, which reveals nothing further on its own.

The remaining issue is stash safety while only part of the new level is active. Our proof has two steps. First, we compare the resizing execution to a reference execution on an infinite-depth tree where resizes have no effect, since it already contains every level. At each time, we project this reference state onto the partially grown shape of the real tree and apply merge-aware post-processing to remove stale copies. Appendix B combines the changing-shape reduction of resizable ORAM [9] with the post-processing of multi-write OSAM [47] to show that the result has the same bucket and stash occupancies as ROGA. Consequently, a stash overflow in ROGA yields an overloaded rooted subtree of the current partially grown shape (Theorem 13). Second, Appendix C bounds this event for every lazy-resize shape. It combines Ring ORAM’s deterministic-eviction tail analysis [51] with multi-write OSAM’s rooted-subtree and multiplicity arguments [47], adapted to keyed routing and aggregation merges. A subtree union bound covers irregular sweep shapes; only the frontier-bucket load remains resize-specific. As detailed in Appendix C, guaranteeing stash safety requires bounding this worst-case frontier load. Specifically, the maximum expected load on any bucket across the sweep is formally bounded by $\mu = \max(A/2, D + \bar{\rho}\tilde{A})$, where $\bar{\rho}$ represents the upper bound on the fraction of fresh keys arriving. To mathematically prevent an overflow, this expected load μ must be kept small enough that the system’s statistical decay rate $q_\rho(t, \mu)$, defined in Corollary 19, strictly exceeds $\ln 4$.

Safe configuration. Our goal is to argue that the stash-overflow probability is negligible. Concretely, we aim for a bound of 2^{-80} and cast our proof of differential obliviousness as conditional on the complementary event that occurs with overwhelming probability $\geq 1 - 2^{-80}$. For the worst case $\rho = 0$, $\bar{\rho} = 1$ identified in Appendix C, $Z = 64$, $A = 20$, $\tilde{A} = 10$, and an early trigger at $D = 25$ keys per leaf give $\mu = 35$, ensuring $q_\rho(t, 35) > \ln 4$ for every stream.

6 Scaling ROGA with Sharding

To parallelize ROGA, we follow Snoopy [22] and split the structure into m independent single-threaded trees, or *shards*, where m is a power of two. The PRF label provides $\log_2 m$ designated bits to select the shard and enough disjoint bits to select a leaf at every supported tree height. Because every copy of a key has the same label, all of its copies reach the same shard and remain mergeable.

Routing updates to their shards as they arrive would expose the data-dependent sequence of shard indices. Updates therefore accumulate in a global batch buffer of capacity B , dis-

tinct from the per-tree overflow stashes, and are routed to all shards only at public flushes. The buffer resides in ordinary observable memory, so the flush pads it to B blocks and uses Snoopy’s oblivious sort–merge–compact–distribute pipeline [22] to send a fixed quota to every shard. Its composite sort key consists of the shard index, dummy flag, and label; one sorting pass therefore both partitions the batch by shard and makes equal keys adjacent. ROGA changes the subsequent merge semantics: instead of Snoopy’s last-write-wins deduplication, it combines duplicate keys with the aggregate operator f from Definition 3. The shards ingest their assigned subarrays in parallel and otherwise retain the single-tree eviction schedule, switching from A to $\tilde{A}$ during a resize sweep.

Shard quota. Every shard processes exactly Q positions, including dummy padding, so its accesses do not reveal its actual load. In the worst case, all B updates are distinct, and the number routed to shard s is $X_s \sim \text{Bin}(B, 1/m)$. For security parameter λ , we choose the smallest Q such that $\Pr(X_s \geq Q) \leq 2^{-\lambda}/m$; a union bound limits quota overflow in any flush to $2^{-\lambda}$. We set $\lambda = 80$ and compute Q from the exact binomial tail, dividing the same failure budget across worker–shard pairs when routing is parallelized.

Coordinated shard resizing. A single coordinator runs Algorithm 3’s estimator; a positive decision doubles all shards in parallel, preserving equal capacities while spending privacy budget only once across shards.

Correctness additionally requires the coordinator to trigger before any other shard overflows. Let d_{tot} be the total number of resident distinct keys and Y_s the load of shard s . The PRF assignment gives $Y_s \sim \text{Bin}(d_{\text{tot}}, 1/m)$ with mean $\mu = d_{\text{tot}}/m$. For $\gamma = 2^{-\lambda}$, a two-sided Chernoff bound and a union bound over all shards show that, while $\mu \leq N$, every shard lies within $\sqrt{3N \ln(2m/\gamma)}$ of the common mean except with probability γ . Let Δ_m be twice this deviation. Thus, on this event no shard exceeds the coordinator by more than Δ_m . We therefore lower the coordinator’s effective load target to $N - \Delta_m$. The not-too-late guarantee of Lemma 9 then initiates resizing before the coordinator reaches that target, leaving enough slack for every other shard to remain below N , except for the estimator and load-balancing failure events. The correction is asymptotically smaller than the $O(N^{2/3})$ headroom of Corollary 11. For concrete parameters we compute a tighter Δ_m from the exact two-sided binomial tails at startup; for $\lambda = 80$, $m = 32$, and $N = 2^{20}$, it is 21,393, or less than 2.1% of per-shard capacity.

7 Implementation

This section describes our Rust implementation of ROGA and how we verify data obliviousness in its security-critical compiled kernels.

7.1 System Implementation

We implement ROGA as a Rust library of roughly 2,500 lines of code. It exposes the interface of Algorithm 1, with the bucket size Z as a compile-time constant and the ordinary and sweep eviction intervals A and $\tilde{A}$ as runtime parameters.

To support multi-core scaling, the library implements the sharded variant described in Section 6. We use the rayon [58] library to process shards in parallel during batch flushes, which

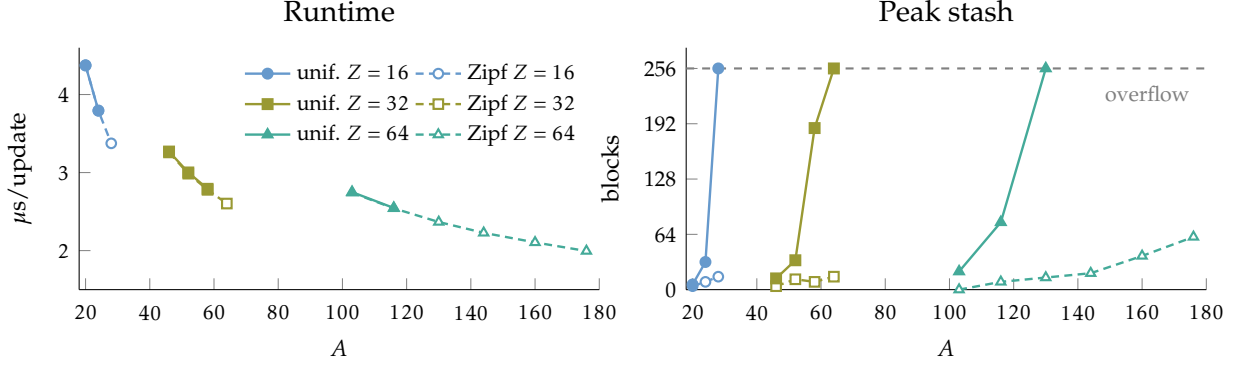


Figure 1: Update cost (left) and peak stash occupancy (right) across eviction intervals A and bucket capacities Z .

run each shard’s evictions, and during resize operations. We implement the constant-time sorting required for batch processing using djsort [12]. The eviction pipeline adapts Snoopy’s sort-merge-compact-distribute approach [22] to tree eviction. In our fixed-width-key setting, the implementation instantiates F as AES-128 CBC-MAC, zero-padding the final input block, and truncates its output to 64 bits. Keys may be of any fixed width, chosen when the structure is initialized, and are 64 bits in our experiments. Each block stores both the resulting 64-bit routing tag $b = F_K(x)$ and the original fixed-width key x . The low bits of b route the block to a leaf, designated upper bits select its shard, and the remaining tag bits provide the secondary ordering used during eviction. The prototype reserves the low 32 bits for leaf routing and caps the tree height at $L = 31$, which suffices for our experiments; taller trees require a wider leaf field or routing tag. Storing b trades 8 bytes per block for one MAC evaluation per access, but the merge does not treat this truncated tag as the block identity: its constant-time equality predicate compares the stored original keys. ROGA’s online merging and occupancy guarantees hold except when two distinct keys receive the same comparison label. The prototype uses an effective 63-bit PRF-derived comparison label, giving collision probability at most $M(M-1)/2^{64}$ for M distinct keys. A collision cannot merge different keys because blocks retain their original keys, but it may prevent all copies of one key from becoming adjacent during eviction. For the 59.4 million distinct keys in our largest experiment, the bound is below 0.02%. All reported executions completed without overflow and produced the correct bulk readout. Deployments requiring negligible collision probability can use wider labels, such as 128 bits. Dummy blocks likewise carry the reserved key $\perp$ in the construction, while the implementation marks them with a status bit.

Because no memory is private in our threat model (Section 2.2), all secret-dependent data movement uses masked conditional moves, with plain branches only on public configuration; the next subsection verifies this property on the compiled binary. In our experiments, the per-tree stash has a fixed capacity of $R = 256$ blocks, and exceeding it aborts ROGA’s execution. Our code does not encrypt data written to external memory because the CVM performs that encryption.

For sharded resizing, only the coordinator runs the load estimator, so the privacy budget is spent once, not per shard.

7.2 Binary Verification of Data Obliviousness

We model an execution’s side-channel leakage as its sequence of branch targets and memory addresses. Data obliviousness in this model is a trace-noninterference property: two executions that agree on public parameters must produce the same leakage trace even when all secret inputs differ. We use BINSEC/REL 0.11.2 to verify this relational property directly on ROGA’s compiled security-critical kernels [21, 31, 56]. For each part of ROGA we test, secret inputs remain symbolic, so a safe result covers every value they can take rather than a sampled set of executions. The fixed size bounds symbolic execution, not the secret-input space.

The targets cover ROGA’s constant-time primitives, stash insertion with buffering (Update), sorting-network aggregation with duplicate reduction (SortAndMerge, Algorithm 2), oblivious bucket placement and eviction (EvictPath), shard load counting and distribution (Section 6), and the computation of the private resize decision, including leaf-count sampling and the fixed-work discrete-Laplace sampler core. The compiler may unroll or vectorize differently for each compile-time bucket size Z , so we verify every evaluated value. Across 23 targets covering 63,288 unrolled instructions, symbolic execution finds no leaks in under 26 seconds.

Analyzing an entire long-running instance symbolically is intractable, so we compose these binary-level results with the structure of the full execution. ROGA invokes the verified kernels in an order and for loop bounds determined by public parameters, including tree height, bucket size, and batch size; larger instances repeat the same trace-noninterfering operations under different public bounds. The sole intentional trace divergence is the final branch on the noisy estimator output, which determines whether the tree resizes and reveals exactly the differentially private decision analyzed in Section 5.

8 Evaluation

We evaluate parameter selection, fixed-capacity performance, multi-core scaling, and private resizing.

Experiments use release builds in an Intel TDX VM on an Intel Xeon 6736P host with 36 physical cores (72 logical cores) and 128 GB of RAM; power-of-two sharding permits up to 64 cores. We disable frequency scaling and turbo boost and pin workers to physical cores where possible.

We report means and standard deviations over three random seeds; coefficients of variation are below 1.1%, except 2.3% for 32-core ROGA. Fixed-capacity runs use an untimed 10,000-operation warmup.

Except in the parameter sweep, experiments use $Z = 64$, $A = 20$, and stash capacity $R = 256$; resizing also uses $\tilde{A} = 10$, as in the safe configuration of Section 5.4. Multicore batches contain $1000 \times m$ real updates padded to $B = 4096 \times m$; the implementation computes Q for $\lambda = 80$ as in Section 6. Keys are 64-bit and sampled uniformly or from a Zipf distribution as stated; n uniform draws from a domain of size U contain $U(1 - (1 - 1/U)^n)$ distinct keys in expectation.

Table 1: Amortized operation cost across oblivious stores.

System	Test Env.	Resize	Block capacity	Cost/op
ZeroTrace [53]	SGX	✗	$\sim 2^{23}$	1218.9 μs
Obliv [44]	SGX	✗	2^{24}	5400 μs
EnigMap [59]	SGX	✗	2^{24}	692.48 μs
Snoopy [22]	SGX (distr.)	✗	$\sim 2^{21}$	10.9 $\mu\text{s}^\dagger$
H2O2RAM [68]	TDX	✗	2^{24}	166.69 μs^*
Obliviouslabs [46]	TDX	✗	2^{24}	141.50 μs^*
BOLT [33]	FPGA/HBM	✗	$\sim 2^{20}$	$\sim 5.7 \mu\text{s}$
Ours	TDX	✓	2^{24}	15.55 μs^*

* Measured by us on a single core of the same machine (block capacity 2^{24}); our row uses the resize-safe parameters $Z = 64$ and $A = 20$. The remaining numbers are taken from the respective papers. † Aggregate throughput of an 18-machine deployment.

Our non-oblivious baseline is Rust’s standard HashMap with its default SipHash-1-3 hasher [1, 8], applying the same merge-by-addition update. Our oblivious baselines are H2O2RAM [68] and Obliviouslabs [46], general doubly oblivious maps supporting arbitrary read–write interleavings. We use H2O2RAM’s public artifact with author-confirmed fixes; it passes our correctness tests in every reported configuration.

Table 1 summarizes the single-core cross-system comparison at block capacity 2^{24} .

8.1 Parameter Sweep

Larger eviction intervals A amortize path accesses but leave more blocks waiting in buckets of size Z . For distinct-key streams, Appendix C admits $A \leq 20, 46$, and 103 for $Z = 16, 32$, and 64 , respectively. Figure 1 sweeps beyond these limits at capacity 2^{18} over 2 million updates. At the admitted limits, uniform workloads peak at only 6, 13, and 21 stash blocks; beyond them, cost falls from $4.37 \mu\text{s}$ at $(Z=16, A=20)$ to $2.55 \mu\text{s}$ at $(Z=64, A=116)$ before overflows occur at $A = 28, 64$, and 130 , respectively. Zipf inputs produce fewer distinct keys and complete all tested intervals through $A = 176$, but relying on this skew would make safety distribution-dependent. We therefore use $(Z=64, A=20)$, where resize safety rather than stash capacity is binding; fixed-capacity deployments can instead select a point on the measured throughput frontier.

8.2 Fixed-Capacity Baseline

Figure 2 compares single-core update costs from block capacity $Z2^L = 2^{16}$ to 2^{26} , using uniform keys from a domain. Each run inserts half the domain into an empty store. When H2O2RAM’s write buffer fills, it compacts the occupied lower levels and rebuilds the next hash-table level; timing the full insertion sequence amortizes these periodic costs, which a short steady-state window could omit. ROGA grows from $8.61 \mu\text{s}$ to $17.49 \mu\text{s}$ per update (up to

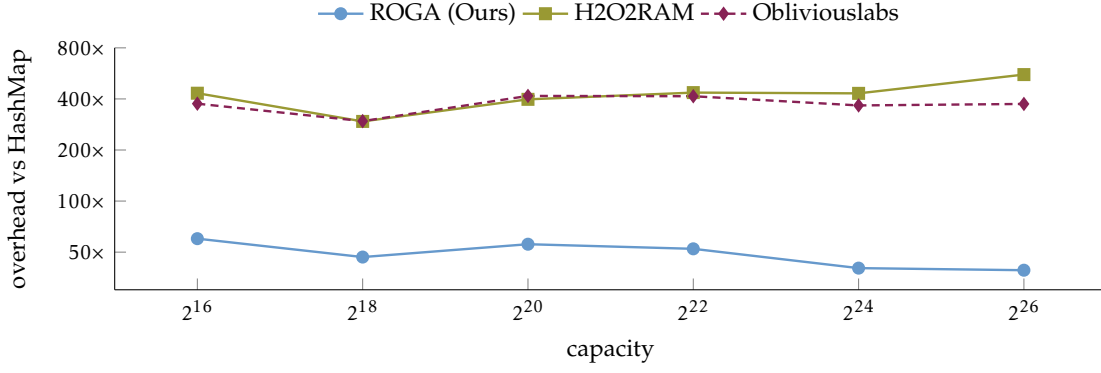


Figure 2: Single-core update-cost overhead from capacity 2^{16} to 2^{26} under uniform updates

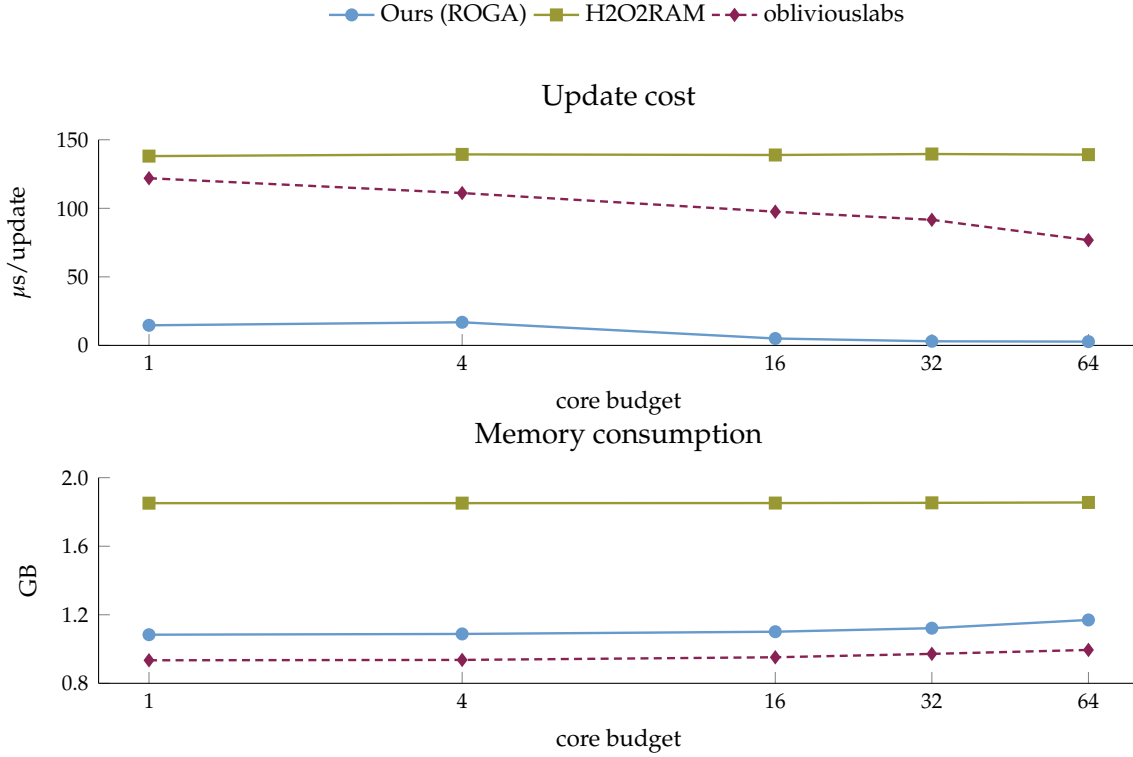


Figure 3: Fixed-capacity scaling for 2^{21} uniform updates on a 2^{22} -key domain. Top: update cost; bottom: peak memory.

67M entries and 8.0 GB). The capacity-matched HashMap cost grows from $0.143 \mu s$ to $0.447 \mu s$, putting ROGA's overhead between $39.1\times$ and $60.0\times$ across the measured range. The oblivious baselines are an order of magnitude slower: Obliviouslabs [46] grows from $53.8 \mu s$ to $167.0 \mu s$, while H2O2RAM grows from $62.0 \mu s$ to $248.8 \mu s$. Our H2O2RAM measurement at block capacity 2^{22} is close to its published single-thread result ($114.22 \mu s$ versus $112.2 \mu s$) [68]. The gap widens with capacity because Obliviouslabs traverses a recursive position map and H2O2RAM periodically rebuilds oblivious hash tables, whereas ROGA needs neither and performs one

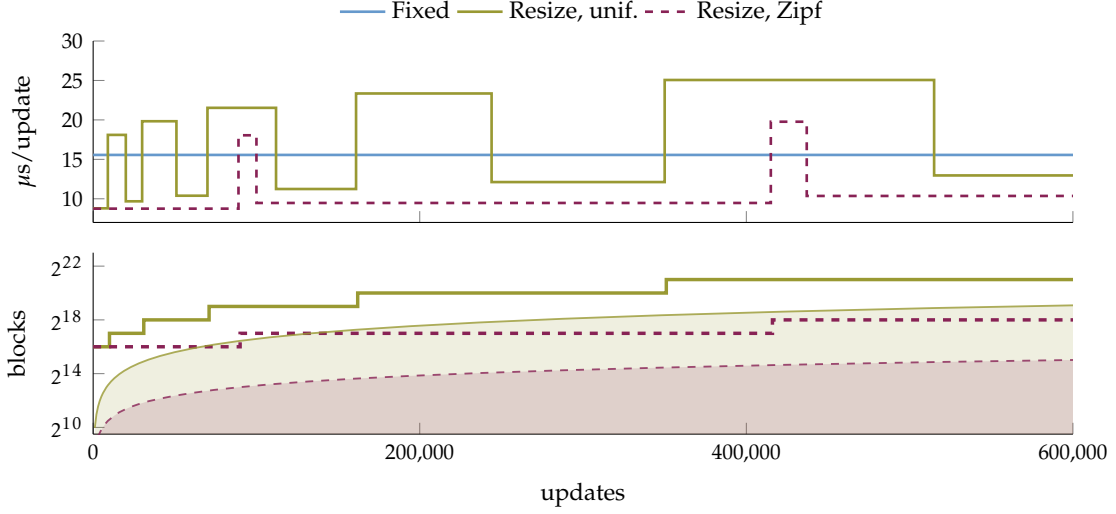


Figure 4: One deterministic resizing trial under uniform and Zipf inputs. Top: update cost compared with fixed provisioning; high plateaus average activation sweeps and low ones steady execution. Bottom: allocated capacity (thick steps) and observed distinct-key count (thin shaded curves).

sort per eviction (Section 4).

8.3 Multi-Core Scaling

Figure 3 compares fixed-capacity scaling over 2^{21} uniform updates from a 2^{22} -key domain at block capacity 2^{23} (1.1 GB). ROGA improves 5.3 $\times$, from 14.66 μs on one core to 2.76 μs on 64 logical cores, reaching within 9.5 $\times$ of the non-oblivious hash map. H2O2RAM remains nearly flat because only its rebuilds parallelize; Obliviouslabs improves 1.6 $\times$ before serial work dominates. At 64 logical cores, ROGA is over 50 $\times$ faster than H2O2RAM and over 27 $\times$ faster than Obliviouslabs and its memory usage is below H2O2RAM’s consistently.

8.4 Resizing

Figure 4 illustrates the first 6×10^5 uniform or Zipf ($s = 1.3$) updates over a 4M-key domain; the comparison below uses all 2 million updates. The fixed baseline provisions 2^{23} blocks for the 4M-key domain. Private resizing starts at 2^{16} with $\alpha = 0.05$ and $\varepsilon = 1$; by Theorem 7, we achieve 1.5-DP. All reported resizing runs completed without an overflow or a failed correctness check. Each plateau in the top panel aggregates the clean windows in one consecutive execution state. Windows contain at most 1,000 updates; we omit only their measured cost at boundaries that mix steady execution with synchronous allocation. The aggregate costs below include allocation, every sweep, and all other resizing work. Pinning the sweep window active for an entire run, an upper bound on its real share, raises per-update cost by 93.6%; ordinary executions pay this higher eviction rate only while activating a new level. Under uniform inputs, resizing doubles seven times to 2^{23} . It therefore has the same 1.08 GB peak as the 2^{23} fixed

baseline, while running $1.32\times$ slower than that baseline and $1.41\times$ slower than an oracle-sized fixed instance. Under Zipf inputs, it reaches 2^{19} with a 77.5 MB peak: $1.36\times$ faster and $14\times$ smaller than fixed provisioning, but 5.3% slower than the oracle-sized instance.

9 Case Study: Oblivious Flow Monitoring

Network operators aggregate per-flow statistics, but revealing the updated grouping keys exposes communication relationships, scans, and access to sensitive services. A fixed-capacity ORAM can hide these accesses only by provisioning for a public flow bound; extrapolating from Section 8.2, capacity 2^{32} would require roughly 512 GB. We evaluate whether private resizing can instead reduce provisioning while allocating for the observed workload.

Setup. We replay the 15-minute MAWI backbone trace collected at samplepoint-F on August 19, 2026 [18, 39]: 277 million valid IP packets across 59.4 million distinct directed source-destination pairs, with a rank-frequency Zipf fit of $s = 1.00$. ROGA represents each pair by a 128-bit PRF fingerprint and uses constant-time merges to aggregate *tshark*’s packet and byte counts and timestamps into a 32-byte record. The run resizes from 2^{16} to 2^{29} blocks using $m = 16$ shards and batch-buffer capacity $B = 4096$. The implementation splits each flush across four routing workers, each with a fixed 1024-block sub-batch and per-shard quota 162 after union-bounding over all worker-shard pairs, for a total per-shard quota of $Q = 4 \cdot 162 = 648$. This is the largest shard count fitting the 128 GB host; the run peaks at 91.3 GB. We measure replay and end-to-end time including PCAP parsing.

Results. Across 16 shards, ROGA replays all packets in 985.3 seconds with peak memory consumption of 91.3 GB. Parsing the PCAP takes another 156.1 seconds, for 1,141.4 seconds end to end. For context, Argus [49] takes 776.3 s, *softflowd* [43] takes 253.3 s on IPv4 packets, and a non-oblivious *HashMap* ingests the pre-parsed records in 58.9 s. Thus, the oblivious aggregation itself has $985.3/58.9 = 16.7\times$ overhead over the non-oblivious aggregation. Including the common 156.1-second parsing step gives a 215.0-second non-oblivious pipeline and $5.3\times$ end-to-end overhead for ROGA. These faster tools do not hide accesses; ROGA’s readout matches the reference across all 59.4 million flows.

10 Related Work

Private analytics in TEEs. Private analytics systems use CVMs and secure enclaves to protect data during processing, often with differentially private outputs [28, 17, 55, 2, 14]. Still, CFC side-channel work and SNPeek show that execution metadata can remain visible [11, 67].

DP for execution leakage. Several systems and definitions use DP to bound leakage based on execution behavior. Hermetic applies DP to timing and message-size side channels in analytics [65]. DP-ORAM and differential obliviousness relax full obliviousness by requiring access patterns to be differentially private with respect to neighboring inputs [61, 15]. Mazloom and Gordon formalize differential obliviousness [40]. ShrinkWrap and Epsolute instantiate it for query processing, padding intermediate tables and per-query result volumes to differentially private sizes over static data [10, 14]. Doquet applies differential obliviousness to range and

join queries over private data structures in a TEE, hiding both access patterns and intermediate result volumes [48]. Menhir combines ORAM-style access hiding with DP-protected volume patterns in an oblivious TEE database [50]. These systems keep storage at a fixed public size and sanitize per-query volume or timing; ROGA instead grows under adversarial insertions.

ORAM, OSAM, and resizing. Path ORAM and Ring ORAM are standard practical baselines for hiding memory access patterns [57, 51]. In the enclave setting, practical oblivious maps include EnigMap [59], the Obliviouslabs oblivious map [46], a Circuit ORAM-based key-value store that builds on the EnigMap line of work, and hierarchical H2O2RAM [68]; the latter two serve as the performance baselines in our evaluation. BOLT moves the oblivious map onto a TEE-enabled FPGA accelerator, and treats sealed high-bandwidth memory as hardware-unobservable storage, cutting the per-access bandwidth overhead to $O(1) + O(\log_2 \log_2 N)$ [33]. OSAM weakens the access model [6] and uses this structure to improve concrete efficiency for workloads in which objects are written and read in a restricted pattern [47]. ROGA builds on OSAM techniques and extends them with merging and resizing for streaming group-by updates followed by scheduled bulk readout. To our knowledge, ROGA is the first doubly oblivious OSAM: it requires no private memory, matching the enclave setting of the doubly oblivious maps it benchmarks against [59, 46, 68].

Private histograms. DP histogram algorithms address privacy concerns for released counts and labels, including sparse-domain and continual-observation settings [25, 52]. These algorithms protect what the analyst receives. Our work is complementary: we protect metadata that is leaked during histogram construction, before the final DP aggregate is released. Additionally, ROGA computes accurate statistics that can later be noised for privacy, whereas other frequency estimators, such as sketches [20, 16], are approximate.

11 Conclusion

We introduced ROGA, a resizable oblivious data structure for exact group-by aggregation in confidential virtual machines. ROGA exploits the structure of aggregation to avoid a recursive position map: keyed routing, single-sort merge-and-placement evictions, and lazy level activation make ingestion without private client memory efficient. An embedded estimator drives a differentially private resize schedule; lazy growth makes it efficient, while our analysis proves differential obliviousness and stash safety. Sharding extends the construction to multicore execution, while binary-level relational verification proves trace noninterference of the compiled security-critical kernels.

These ideas translate into substantial practical gains: ROGA outperforms state-of-the-art oblivious maps across all evaluated capacities, scales to 64 logical cores, reduces memory by up to 14× through private resizing, and processes a 277-million-packet network trace while producing exact aggregates. More broadly, a private control signal lets secure systems adapt observable resources to realized workloads.

ROGA deliberately targets streaming aggregation with scheduled bulk readout because unchanged labels would make point queries reveal repeated lookups. The implementation also aborts on the bounded-probability event of stash overflow; extending private adaptation to online queries and other oblivious data structures remains future work.

Open Science

We release the artifacts of this paper at <https://github.com/google-research/roga>.

References

- [1] Abseil team. Swiss tables design notes. <https://abseil.io/about/design/swisstables>, 2018.
- [2] Joshua Allen, Bolin Ding, Janardhan Kulkarni, Harsha Nori, Olga Ohrimenko, and Sergey Yekhanin. An algorithmic framework for differentially private data analysis on trusted processors. In *Advances in Neural Information Processing Systems*, volume 32, 2019.
- [3] AMD. AMD Secure Encrypted Virtualization (SEV). <https://www.amd.com/en/developer/sev.html>. Accessed 2026-08-25.
- [4] AMD. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more. *White Paper*, 2020.
- [5] AMD. 5th Gen AMD EPYC Processor Architecture. White paper, Publication 70353, 2025.
- [6] Ananya Appan, David Heath, and Ling Ren. Oblivious single access machines: A new model for oblivious computation. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 3080–3094, 2024.
- [7] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In *Advances in Cryptology—Eurocrypt 2020*.
- [8] Jean-Philippe Aumasson and Daniel J. Bernstein. SipHash: A fast short-input PRF. In *Progress in Cryptology—INDOCRYPT 2012, 13th International Conference on Cryptology in India*, volume 7668 of *LNCS*, pages 489–508. Springer, 2012.
- [9] Amine Bahi, Brice Minaud, and Tarik Moataz. Resizable oblivious RAM. In *Advances in Cryptology—Eurocrypt 2026*, *LNCS*, pages 273–302. Springer, 2026.
- [10] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. ShrinkWrap: Efficient SQL query processing in differentially private data federations. *Proceedings of the VLDB Endowment*, 12(3):307–320, 2018.
- [11] James Bell-Clark, Albert Cheu, Adria Gascon, and Jonathan Katz. Hardening confidential federated compute against side-channel attacks. arXiv preprint arXiv:2603.21469, 2026.
- [12] Daniel J. Bernstein. djbSort: Software for software-sorting networks. <https://sorting.cr.yp.to/>, 2018.
- [13] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proc. 26th Symposium on Operating Systems Principles (SOSP)*, pages 441–459, 2017.

- [14] Dmytro Bogatov, Georgios Kellaris, George Kollios, Kobbi Nissim, and Adam O’Neill. ϵ psolute: Efficiently querying databases while providing differential privacy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2262–2276, 2021.
- [15] T.-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. *Journal of the ACM*, 69(4):27:1–27:49, 2022.
- [16] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.
- [17] Albert Cheu, Artem Lagzdin, Brett McLarnon, Daniel Ramage, Katharine Daly, Marco Gruteser, Peter Kairouz, Rakshita Tandon, Stanislav Chiknavaryan, Timon Van Overveldt, and Zoe Gong. Toward provably private analytics and insights into GenAI use. arXiv preprint arXiv:2510.21684, 2025.
- [18] Kenjiro Cho, Koushirou Mitsuya, and Akira Kato. Traffic data repository at the WIDE project. In *USENIX Annual Technical Conference, FREENIX Track*, 2000.
- [19] Jalen Chuang, Alex Seto, Nicolas Berrios, Stephan van Schaik, Christina Garman, and Daniel Genkin. TEE.fail: Breaking trusted execution environments via DDR5 memory bus interposition. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2026.
- [20] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.
- [21] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level. In *IEEE Symposium on Security and Privacy*, pages 1021–1038, 2020.
- [22] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proc. ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP)*, pages 655–671, 2021.
- [23] Ankur Dave, Chester Leung, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Oblivious cooperative analytics using hardware enclaves. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys)*, pages 39:1–39:17, 2020.
- [24] Sen Deng, Mengyuan Li, Yining Tang, Shuai Wang, Shoumeng Yan, and Yinqian Zhang. CipherH: Automated detection of ciphertext side-channel vulnerabilities in cryptographic implementations. In *USENIX Security Symposium*, 2023.
- [25] Damien Desfontaines, James Voss, Bryant Gipson, and Chinmoy Mandayam. Differentially private partition selection. *Proceedings on Privacy Enhancing Technologies*, 2022(1):339–352, 2022.
- [26] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. 3rd Conf. on Theory of Cryptography (TCC)*, pages 265–284, 2006.

- [27] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.
- [28] Hubert Eichner, Daniel Ramage, Kallista Bonawitz, Dzmitry Huba, Tiziano Santoro, Brett McLarnon, Timon Van Overveldt, Nova Fallen, Peter Kairouz, Albert Cheu, Katharine Daly, Adria Gascon, Marco Gruteser, and Brendan McMahan. Confidential federated computations. arXiv preprint arXiv:2404.10764, 2024.
- [29] Saba Eskandarian and Matei Zaharia. ObliDB: Oblivious query processing for secure databases. *Proceedings of the VLDB Endowment*, 13(2):169–183, 2019.
- [30] Craig Gentry, Kenny A. Goldman, Shai Halevi, Charanjit S. Jutla, Mariana Raykova, and Daniel Wichs. Optimizing ORAM and using it efficiently for secure computation. In *Privacy Enhancing Technologies Symposium (PETS) 2013*, volume 7981 of LNCS, pages 1–18. Springer, 2013.
- [31] Lukas Gerlach, Robert Pietsch, and Michael Schwarz. Do compilers break constant-time guarantees? In *Financial Cryptography and Data Security (FC)*, pages 327–344, 2025.
- [32] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *JACM*, 1996.
- [33] Yitong Guo, Hongbo Chen, Haobin Hiroki Chen, Yukui Luo, XiaoFeng Wang, and Chenghong Wang. BOLT: Bandwidth-optimized lightning-fast oblivious map powered by secure HBM accelerators. In *Proc. 2025 ACM Conference on Computer and Communications Security (CCS)*, 2025.
- [34] Intel. Intel Trust Domain Extensions, 2023.
- [35] Fumiyuki Kato, Yang Cao, and Masatoshi Yoshikawa. Olive: Oblivious federated learning on trusted execution environment against the risk of sparsification. *Proceedings of the VLDB Endowment*, 16(10):2404–2417, 2023.
- [36] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A systematic look at ciphertext side channels on AMD SEV-SNP. In *S&P*, 2022.
- [37] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHER-LEAKS: Breaking constant-time cryptography on AMD SEV via the ciphertext side channel. In *USENIX Security*, 2021.
- [38] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer Science & Business Media, 2007.
- [39] MAWI Working Group. Traffic Trace at Samplepoint-F, August 19, 2026, 14:00–14:15. WIDE Project Traffic Archive, 2026.

- [40] Sahar Mazloom and S. Dov Gordon. Secure computation with differentially private access patterns. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 490–507, 2018.
- [41] Frank D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *SIGMOD*, 2009.
- [42] Meta Platforms. Path ORAM rust implementation. <https://github.com/facebook/oram>, 2024.
- [43] Damien Miller and Hitoshi Irino. softflowd: Flow-based network traffic analyser. <https://github.com/irino/softflowd/releases/tag/softflowd-v1.1.0>, 2022. Version 1.1.0.
- [44] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 279–296, 2018.
- [45] Tarik Moataz, Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Resizable tree-based oblivious RAM. In *Financial Cryptography and Data Security—FC 2015*, volume 8975 of *LNCS*, pages 147–167. Springer, 2015.
- [46] Oblivious Labs Inc. Oblivious map: An oblivious key-value store based on circuit ORAM. <https://github.com/obliviouslabs/oram>. Accessed 2026-07-22.
- [47] Sage Pia, Ananya Appan, Maryam Rezapour, Amey Shukla, Nikhil Date, Benjamin Fuller, Ling Ren, and David Heath. Oblivious single access machines are concretely efficient. <https://eprint.iacr.org/2026/451>, 2026. IACR Cryptology ePrint Archive, Report 2026/451.
- [48] Lina Qiu, Georgios Kellaris, Nikos Mamoulis, Kobbi Nissim, and George Kollios. Doquet: Differentially oblivious range and join queries with private data structures. *Proceedings of the VLDB Endowment*, 16(13):4160–4173, 2023.
- [49] QoSient, LLC. Argus: Audit record generation and utilization system. <https://qosient.com/argus/>, 2024.
- [50] Leonie Reichert, Gowri R Chandran, Phillipp Schoppmann, Thomas Schneider, and Björn Scheuermann. Menhir: An oblivious database with protection against access and volume pattern leakage. In *AsiaCCS*, pages 1675–1690, 2024.
- [51] Ling Ren, Christopher Fletcher, Albert Kwon, Emil Stefanov, Elaine Shi, Marten Van Dijk, and Srinivas Devadas. Constants count: Practical improvements to oblivious RAM. In *24th USENIX Security Symposium*, pages 415–430, 2015.
- [52] Adrian Rivera Cardoso and Ryan Rogers. Differentially private histograms under continual observation: Streaming selection into the unknown. In *Proceedings of the 25th International Conference on Artificial Intelligence and Statistics*, volume 151 of *Proceedings of Machine Learning Research*, pages 2397–2419, 2022.

- [53] Sajin Sasy, Sergey Gorbunov, and Christopher W. Fletcher. ZeroTrace: Oblivious memory primitives from Intel SGX. In *NDSS*, 2018.
- [54] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Fast fully oblivious compaction and shuffling. In *CCS*, 2022.
- [55] Sajin Sasy and Olga Ohrimenko. Oblivious sampling algorithms for private data analysis. In *Advances in Neural Information Processing Systems*, volume 32, 2019.
- [56] Moritz Schneider, Daniele Lain, Ivan Puddu, Nicolas Dutly, and Srdjan Čapkun. Breaking bad: How compilers break constant-time implementations. In *AsiaCCS*, pages 1690–1706, 2025.
- [57] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: An extremely simple oblivious RAM protocol. *Journal of the ACM (JACM)*, 65(4):1–26, 2018.
- [58] Josh Stone and Niko Matsakis. Rayon: A data parallelism library for Rust. <https://github.com/rayon-rs/rayon>, 2025.
- [59] Afonso Tinoco, Sixiang Gao, and Elaine Shi. EnigMap: External-memory oblivious map for secure enclaves. In *32nd USENIX Security Symposium*, pages 4033–4050, 2023.
- [60] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Workshop on System Software for Trusted Execution*, 2017.
- [61] Sameer Wagh, Paul Cuff, and Prateek Mittal. Differentially private oblivious RAM. *Proceedings on Privacy Enhancing Technologies*, 2018(4):64–84, 2018.
- [62] Luca Wilke, Florian Sieck, and Thomas Eisenbarth. TDXdown: Single-stepping and instruction counting attacks against Intel TDX. In *CCS*, 2024.
- [63] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A single-stepping framework for AMD-SEV. In *CHES*, 2024.
- [64] Royce J. Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. Differentially private SQL with bounded user contribution. *Proc. Priv. Enhancing Technol. (PoPETs)*, 2020(2):230–250, 2020.
- [65] Min Xu, Antonis Papadimitriou, Ariel Feldman, and Andreas Haeberlen. Using differential privacy to efficiently mitigate side channels in distributed analytics. In *EuroSec*, 2018.
- [66] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P*, 2015.
- [67] Ruiyi Zhang, Albert Cheu, Adria Gascon, Daniel Moghimi, Phillipp Schoppmann, Michael Schwarz, and Octavian Suci. SNPeek: Side-channel analysis for privacy applications on confidential VMs. In *NDSS*, 2025.

- [68] Leqian Zheng, Zheng Zhang, Wentao Dong, Yao Zhang, Ye Wu, and Cong Wang. H2O2RAM: A high-performance hierarchical doubly oblivious RAM. In *34th USENIX Security Symposium*, 2025.
- [69] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, 2017.

A Notation

	Symbol	Meaning
Stream	S, n	keyed input stream of n key-value pairs (Definition 2)
	$\mathcal{K}, \mathcal{V}$	key universe and value domain
	$\text{keys}(S)$	set of keys appearing in S
	f	aggregate operator (Definition 3)
Privacy	ϵ	privacy budget of the resize schedule
	Δ	query sensitivity, with $\Delta = 2^L/k$ for the load estimator
	α	failure probability of the sampling bound and of the noise quantile
Structure	N	distinct-key capacity target of one tree, $N = D2^L$
	L	tree height, so the tree has 2^L leaves, following [57, 51]
	Z	bucket size in slots, one block per slot, following [57, 51]
	D	target number of distinct keys per leaf, set to 25 in our resizing experiments
	R	fixed capacity of one tree’s stash in blocks
	A	ordinary and maximum eviction interval, following [51]
	$\tilde{A}$	eviction interval while a sweep runs, with $\tilde{A} \leq A$
	$d_t, d_{e,t}$	distinct keys held after update t , and keys first entering the tree in epoch e by then
	M	total number of distinct keys in the first T updates, so $M = d_T$
	K, F_K, b	PRF key internal to the CVM, the keyed PRF, and the block label $b = F_K(x)$
	e	epoch counter of a block, the epoch in which its key claimed a slot
Sharding	μ	uniform upper bound on expected bucket load before post-processing
	m, s	shard count, a power of two, and a shard index
	B	padded capacity of the sharding batch buffer in blocks
	Q	per-shard block quota for one flush
Estimator	λ	statistical security parameter for the sharding failure bounds, set to 80
	k	evictions per measurement interval, $k = \lfloor (N2^L/A^2)^{1/3} \rfloor$
	$\hat{d}$	load estimate, $\hat{d} = \Delta \sum_{i=1}^k c_i$
	τ	noiseless threshold derived from N
	β	Chernoff deviation of the estimator, $\beta = \sqrt{3N\Delta \ln(1/\alpha)}$
	$\hat{\tau}, \theta$	internal noisy threshold $\hat{\tau} = \tau + \theta - 2\zeta$ and its noise draw θ
	η, ζ	per-measurement noise draw and the $(1 - \alpha)$ -quantile of the noise distribution

B Lazy Resizing Reduction

This appendix reduces lazy resizing to node-by-node growth and shows that stash overflow at any time yields an overloaded subtree of the current partially grown shape. The proof distinguishes *physical allocation* from *logical growth*. At a resize, ROGA reserves memory for the entire new level, but every new bucket is empty and does not participate in an eviction until its activating eviction. Consequently, reserving that memory does not change the ORAM state: until a sibling pair is activated, the execution is identical to one in which that pair does not yet exist. The logical tree therefore grows one sibling pair at a time, at the activating evictions, even though its backing memory was allocated all at once. We only grow the tree, so we do not need the shrinking direction of [9], which drains the removed nodes back into the stash.

Lemma 12 (Capacity arrival). *A new bucket holds no block before the eviction that reaches it, and each eviction reaches exactly two new buckets, so all 2^{L+1} of them become reachable within $2^L \tilde{A}$ insertions.*

Proof. An eviction loads one root-to-leaf path plus one sibling, which is exactly the two children of the node of level L on that path. Those two lie under no other node of level L , and the schedule reaches each of the 2^L nodes of that level once per 2^L evictions. $\square$

Following [9], fix a finite operation prefix s and choose a height H larger than every height reached during that prefix. We compare the real structure against a complete reference tree of height H whose buckets hold arbitrarily many blocks. Every logical shape reached during s is a subtree of this reference tree, which we call the *infinite tree*. The infinite execution $\text{ROGA}^\infty[s]$ runs the same operation sequence s as the real structure, with three differences. Buckets never overflow, so nothing is ever pushed to the stash for lack of room. A physical resize does nothing, because the reference tree already contains every level reached during the prefix. An eviction follows its branch to depth H and writes each block back at the deepest bucket on its own path.

That branch is fixed by the eviction counter, read one bit per level down. The real tree stops after L levels, so the real path is the first L steps of it. Doubling the height reads one more bit and takes one more step down the same branch. An eviction therefore follows the same branch before and after a resize, which is what makes the resize invisible to the infinite execution. We write $\text{ROGA}[s]$ for the state of the real structure after the same sequence.

We first remove the payload semantics from the argument. Give every insertion a monotonically increasing *version*, and define the *overwrite shadow* of a ROGA execution by replacing f with the operator that keeps the newest value. The shadow retains the key, routing label, newest version, and minimum epoch of every merged run. Eviction control and placement inspect only these fields, not the payload, and every nonempty run produces exactly one surviving block. An induction over evictions therefore shows that ROGA and its shadow load the same working sets, collapse the same runs to the same surviving versions, and place those survivors in the same buckets. They therefore have identical bucket occupancies, stash occupancies, estimator inputs, and resize schedule. Only their payload values differ.

It remains to relate the bounded and infinite-capacity overwrite shadows. For each operation index t , define a capacity function Φ_t on the reference tree. It assigns capacity Z to every bucket of the old tree and to every new sibling pair whose activating eviction has completed,

and capacity 0 to every other bucket. Thus, Φ_t describes the logical tree after operation t , independently of which buckets have already been reserved in physical memory. Let R_t and I_t be the bounded and infinite-capacity shadow states at that time.

The post-processing has two parts. First, the shape projection P_{Φ_t} moves every block outside the active logical tree to its deepest active ancestor; active buckets remain unbounded. Second, apply the merge-aware post-processing of [47] to this projected state. Process versions from the leaves to the root, regarding the stash as an ancestor of the root. A version that occurs in the same bucket of R_t stays there; a version that occurs in an ancestor bucket of R_t is moved upward until it reaches that bucket; and a version absent from R_t is deleted as stale. We denote the composition by G_{Φ_t, R_t} . Unlike capacity-only post-processing, this relation allows the bounded execution to have merged copies earlier than the infinite-capacity execution.

Theorem 13 (Lazy resize reduction). *For every time t , $G_{\Phi_t, R_t}(I_t)$ and R_t contain the same versions in every bucket and in the stash. Consequently, if the stash of R_t exceeds R , then some rooted subtree T of the active logical tree contains more than $Z|T| + R$ blocks in $P_{\Phi_t}(I_t)$ before merge-aware post-processing.*

Proof sketch. First replace the lazy execution by the following logically equivalent node-by-node execution. A physical resize is a no-op. When an eviction reaches an inactive sibling pair for the first time, the execution adds that pair to the logical tree and immediately performs the activating eviction, loading and writing back both new children. All other operations are unchanged.

The lazy and node-by-node executions have identical states after every operation. Before a pair is activated, its physical buckets are empty and do not participate in evictions, so omitting them changes neither an eviction's working set nor its write-back; readouts are read-only and do not change this state argument. At activation, both executions add the same two empty buckets and run the same greedy eviction over the same working set. Afterward, the pair belongs to both logical trees. Induction over the operations therefore proves the equivalence.

We now combine the two prior post-processing arguments. Lemma 1 of [9] proves that changing a capacity function and performing the accompanying eviction commute with shape projection. Lemmas 1 and 2 of [47] prove, for a fixed shape, the following one-sided invariant: every version remaining in the bounded execution also occurs in the infinite-capacity execution, in the same bucket or below it. The bounded execution may have removed additional stale versions by merging them earlier. These proofs use an eviction path only through greedy write-back along that path, so neither requires the paths to be random.

Their invariants compose by induction over ROGA's operations. A buffered insertion appends the same fresh version to both stashes and changes no bucket; hence batching a public number of insertions between evictions does not affect the deterministic relation. For an ordinary eviction, the merge-aware step of [47] preserves the one-sided version invariant, and the shape projection is unchanged. For an activating eviction, the capacity change and write-back commute with shape projection by the resize case of [9]; the same eviction preserves the version invariant by the preceding OSAM argument. Indeed, when the infinite-capacity activation removes a stale version, that version is either already absent on the bounded side or lies in the stash or an ancestor bucket on the same branch and is loaded there as well; a merge made only on the bounded side is permitted by the one-sided invariant. In particular, a merge made by the

infinite-capacity execution is also made by the bounded execution, while a merge made only by the bounded execution merely causes the corresponding stale version to be deleted during G_{Φ_t, R_t} . This proves the first claim.

For the second claim, apply Lemma 3 of [47] to the active logical shape. Its maximal-full-subtree proof uses the greedy-write-back property that a surviving stash block cannot have passed an eligible non-full ancestor; deleting stale versions only removes blocks and does not create a new stash survivor. It follows that if more than R versions remain in the stash, the pre-processed occupancy of some rooted subtree exceeds its capacity $Z|T|$ by more than R . The argument uses only ancestry, greedy placement, and bucket capacities, so it also holds for the irregular shapes reached during a sweep. Finally, the overwrite-shadow equivalence transfers both claims from R_t back to ROGA.

C Stash Analysis of the Keyed Multi-Write OSAM

This appendix bounds stash overflow for every tree shape that can occur during the lazy-resizing execution of Appendix B. Such a shape consists of the old perfect tree plus some pairs of sibling buckets on the new level. Together, Appendices B and C combine the changing-shape reduction of resizable ORAM [9], the deterministic-eviction stash analysis of Ring ORAM [51], and techniques from multi-write OSAM [47] for handling correlated copies of repeated keys.

Following the infinite-ORAM framework of Path ORAM and Ring ORAM [57, 51], we first allow every bucket to hold arbitrarily many blocks. With repeated keys, capacity-only post-processing is insufficient, because the bounded execution may merge stale copies before the infinite-capacity execution does. The post-processing of [47] additionally deletes those stale copies and produces the same bucket occupancies and stash as the bounded execution. Appendix B combines that merge-aware relation with the changing-capacity reduction of [9]. As in Lemma 3 of [47], stash overflow then implies that some rooted subtree T holds more blocks than the total capacity assigned to T plus R . This reduction is combinatorial and carries over unchanged.

Ring ORAM’s Chernoff bound assumes that every block has an independent random leaf. ROGA has no position map and does not remap blocks, so every copy of a key keeps the same routing label $F_K(x)$. Labels are independent across distinct keys, but live copies of the same key are correlated. To control this correlation, we follow the multi-write OSAM analysis [47] and bound how many keys have j live copies in each rooted subtree. Fix a rooted subtree T . For $j \geq 1$, let w_j be the number of keys with exactly j live blocks in T . The total load of T is therefore $\sum_j j w_j$. We show that $\mathbb{E}[w_j]$ decreases geometrically with j , which produces the extra repeat term in our bound. These counts factorize in the moment generating function: across keys they are independent, since each key has its own leaf, and for one key the events of holding exactly j blocks are mutually exclusive [47]. We treat the leaves as independent and uniform, which holds up to the security of the PRF that derives them.

Every eviction’s working set contains the entire stash. Under ROGA’s eviction order, a bucket at level ℓ is visited once every 2^ℓ evictions, with successive visits continuing through alternating children. Thus, at most $A 2^\ell$ insertions occur between two consecutive visits to the bucket. Once both children of a bucket have been activated, their parent behaves like an

ordinary internal bucket. Because consecutive visits alternate between its children, any block already present at the previous visit is pushed into a child by the current visit. Thus, only blocks inserted since the previous visit can remain in the parent. Each such block lands there with probability $2^{-(\ell+1)}$, so the bucket contains at most $A/2$ blocks in expectation. A block may merge several insertions, but they all occurred since the previous visit, which pushed every older copy of its key strictly below the bucket.

Two structural quantities drive the bound before any tail inequality is applied. A fixed subtree of n buckets has capacity Zn and expected load at most μn . Repeated keys add correlation: the proof below shows that the expected number of keys with j live copies decreases geometrically as 2^{-j} . The theorem turns these inputs into a probability bound by introducing a free Chernoff parameter.

Theorem 14 (Keyed multi-write stash bound). *Run ROGA on a fixed tree shape reached during resizing with bucket size Z , and suppose every bucket contains at most μ blocks in expectation before post-processing. For every free Chernoff parameter $0 < t < \ln 2$, define the duplicate penalty and the fixed-subtree decay rate by*

$$\text{dup}(t) = \sum_{j \geq 2} ((e^{tj} - 1) - j(e^t - 1))2^{-j} = \frac{2(e^t - 1)^2}{2 - e^t},$$

$$q(t, \mu) = tZ - (e^t - 1)\mu - 3A \text{dup}(t).$$

If $q(t, \mu) > \ln 4$, then

$$\Pr[\text{stash} > R] \leq \frac{e^{A \text{dup}(t) - tR}}{1 - 4e^{-q(t, \mu)}}.$$

The restriction $t < \ln 2$ makes the duplicate-penalty series converge. The duplicate penalty is the extra exponential-moment cost of correlated copies, while $q(t, \mu)$ is the resulting decay per bucket for one fixed subtree. For one fixed subtree of n buckets, the bound decays as $e^{-nq(t, \mu)}$. There are at most 4^n rooted binary subtrees of size n , so the union bound and the sum over subtree sizes form a geometric series with ratio $4e^{-q(t, \mu)}$; the condition $q(t, \mu) > \ln 4$ makes this series converge. The parameter t is chosen only afterward to optimize the bound and is not part of ROGA.

The proof uses two structural steps. If T contains n buckets, its expected total load is at most μn by the load assumption. For each $j \geq 2$, we must bound the expected number of keys that have exactly j live copies inside T at the analyzed time. Conceptually label each block by the newest insertion merged into it. The eviction order keeps these labels in insertion order along the key's path, even when blocks merge. Consequently, maintaining j live copies requires the chain to span j distinct buckets, which makes the geometric multiplicity count possible. We first establish this insertion-ordering invariant, extending the analysis of [47] to account for ROGA's merge operation.

Lemma 15 (Insertion ordering). *The tree-resident blocks of one key occupy distinct buckets on the path to its leaf. Along this path, a block with a newer insertion label is strictly closer to the root than one with an older label.*

Proof. By induction over evictions. A new insertion starts in the stash, above every tree bucket. During an eviction, all same-key blocks in the working set merge, and the result carries their newest insertion label. Suppose this result were deposited at or below a tree-resident block with an older label that was not loaded. Both blocks follow the same path, so the older block's bucket would lie on the evicted path and would have been loaded, a contradiction. The merge therefore preserves insertion order, and same-key blocks loaded from one bucket merge before write-back, preserving distinct buckets as well. $\square$

The invariant lets us extend the multiplicity analysis of [47] to ROGA. The case $j = 1$ is already covered by the expected-load bound μn ; only $j \geq 2$ represents duplicates and guarantees that the newest copy has an older live sibling.

Lemma 16 (Multiplicity decay). *Fix a rooted subtree T with n buckets. For $j \geq 2$, let w_j be the number of keys with exactly j live blocks in T . Then $\mathbb{E}[w_j] \leq A(3n+1)2^{-j}$.*

Proof. We adapt the counting argument of [47], tracing the changes caused by ROGA's eviction interval and merge rule. Their proof counts, for each bucket b of T at level ℓ , the sequences of j blocks of one key that end in b . By the insertion ordering of Lemma 15, such a sequence occupies distinct buckets of one path, so it starts at a level $\ell' \leq \ell - j + 1$, and a candidate remains in b only if its path diverges from the evicted path at b , which happens with probability $2^{-(\ell+1)}$. The candidates at the start level are the blocks arriving between the evictions of that bucket's two children. [47] evict once per access, so this interval holds $2^{\ell'}$ candidates, while ROGA waits at most A insertions between evictions, so the same interval spans at most $A2^{\ell'}$ insertions. Moreover, the start bucket holds the sequence's shallowest block, by Lemma 15 its newest one, and for $j \geq 2$ that block belongs to a key with an older live block, so its insertion carried a repeated key. Bounding repeated-key insertions by all insertions, the interval contributes at most $A2^{\ell'}$ candidates. Because $\ell' \leq \ell - j + 1$, multiplying this candidate count by the survival probability gives

$$A2^{\ell'}2^{-(\ell+1)} \leq A2^{-j}.$$

The remaining bucket and boundary count is unchanged from [47] and contributes the factor $3n+1$, giving the stated bound. $\square$

Proof of Theorem 14. Fix a rooted subtree T with n buckets. In the moment generating function, split the contribution of multiplicity j as

$$e^{tj} - 1 = j(e^t - 1) + ((e^{tj} - 1) - j(e^t - 1)).$$

The linear part is covered by the expected-load bound μn , while Lemma 16 bounds the remaining convexity premium by $\text{dup}(t)$. The $3n$ term of that lemma enters the fixed-subtree decay rate, and its $+1$ term enters the prefactor, giving

$$\Pr[T \text{ holds more than } Zn + R \text{ blocks}] \leq e^{A\text{dup}(t) - tR} e^{-nq(t, \mu)}.$$

There are at most 4^n rooted binary subtrees of size n . Taking a union bound and summing over n gives a geometric series with ratio $4e^{-q(t, \mu)}$, which converges when $q(t, \mu) > \ln 4$ and yields the claimed bound. $\square$

Tightening the duplicate penalty. Theorem 14 is already complete, but its closed-form duplicate penalty allows every multiplicity bound of Lemma 16 to be tight simultaneously. The total number of surplus copies gives an additional constraint that produces a tighter bound.

Duplicates admit this global budget because lifetime and landing probability cancel level by level: a surplus copy parks at level ℓ with probability $2^{-(\ell+1)}$ and is collected within the 2^ℓ evictions until its bucket's next visit, so every bucket holds the same expected amount of duplicate inventory.

Lemma 17 (Repeat budget). *Fix a rooted subtree T with n buckets. For $j \geq 2$, let w_j be the number of keys with exactly j live blocks in T . Then $\sum_{j \geq 2} (j-1) \mathbb{E}[w_j] \leq A n/2$.*

Proof. A key with exactly j live blocks in T has $j-1$ blocks that are not its deepest one, so the left side counts the non-deepest live blocks in T . Map each such block to the newest insertion merged into it. The blocks of one key partition its insertions, so the map is injective, and the witness insertion carries a repeated key, since the block has an older live sibling. A block in a bucket at level ℓ contains only insertions since that bucket's previous visit, so the witness is one of at most $A 2^\ell$ insertions, and its block resides in the bucket with probability $2^{-(\ell+1)}$. Summing $A 2^\ell 2^{-(\ell+1)} = A/2$ over the n buckets gives the bound. $\square$

Corollary 18 (Budget-refined stash bound). *For $0 < t < \ln 2$, define*

$$\text{dup}_\star(t) = \max_{\{u_j\}_{j \geq 2}} \sum_{j \geq 2} ((e^{tj} - 1) - j(e^t - 1)) u_j$$

subject to

$$0 \leq u_j \leq 2^{-j}, \quad \sum_{j \geq 2} (j-1) u_j \leq \frac{1}{6}.$$

The maximum is attained by $u_6 = 11/960$, $u_j = 2^{-j}$ for $j \geq 7$, and $u_j = 0$ otherwise. Hence, writing $x = e^t$,

$$\text{dup}_\star(t) = \frac{11}{960} (x^6 - 6x + 5) + \sum_{j \geq 7} \frac{x^j - jx + j - 1}{2^j}.$$

The bound of Theorem 14 remains valid when $\text{dup}(t)$ is replaced throughout by $\text{dup}_\star(t)$.

Proof. Let $d_j = (e^{tj} - 1) - j(e^t - 1)$ be the penalty contributed by multiplicity j . The benefit per unit of the repeat budget is

$$\frac{d_j}{j-1} = (x-1) \frac{1}{j-1} \sum_{k=1}^{j-1} (x^k - 1),$$

which increases with j . The maximizing sequence therefore sets $u_j = 2^{-j}$ for $j \geq 7$, uses the remaining budget at $j = 6$, and sets the lower coordinates to zero. Since

$$\sum_{j \geq 7} (j-1) 2^{-j} = \frac{7}{64},$$

the remaining budget gives $u_6 = 11/960$. Substituting these values into the objective yields the displayed expression.

Fix a rooted subtree T with n buckets and normalize its expected multiplicity counts as

$$u_j = \frac{\mathbb{E}[w_j]}{A(3n+1)}.$$

Lemma 16 gives $0 \leq u_j \leq 2^{-j}$, while Lemma 17 gives

$$\sum_{j \geq 2} (j-1)u_j \leq \frac{n}{2(3n+1)} \leq \frac{1}{6}.$$

Thus, $(u_j)_{j \geq 2}$ is feasible in the maximization that defines $\text{dup}_\star(t)$, and hence

$$\begin{aligned} \sum_{j \geq 2} ((e^{tj} - 1) - j(e^t - 1)) \mathbb{E}[w_j] \\ \leq A(3n+1) \text{dup}_\star(t) = 3An \text{dup}_\star(t) + A \text{dup}_\star(t). \end{aligned}$$

The first term replaces the duplicate contribution in the fixed-subtree decay rate, and the second replaces the duplicate contribution in the prefactor of Theorem 14. $\square$

Corollary 19 (Repeat-aware stash bound). *Under the conditions of Theorem 14, suppose at most a $(1 - \rho)$ fraction of the insertions between any two consecutive evictions through the same bucket carry a repeated key. For $0 < t < \ln 2$, define the refined fixed-subtree decay rate*

$$q_\rho(t, \mu) = tZ - (e^t - 1)\mu - 3(1 - \rho)A \text{dup}(t).$$

If $q_\rho(t, \mu) > \ln 4$, then

$$\Pr[\text{stash} > R] \leq \frac{e^{(1-\rho)A \text{dup}(t) - tR}}{1 - 4e^{-q_\rho(t, \mu)}}.$$

The same bound holds with $\text{dup}_\star(t)$ in place of $\text{dup}(t)$.

Proof. Only repeated-key insertions contribute candidates in the proofs of Lemmas 16 and 17. The assumption therefore multiplies both bounds by $(1 - \rho)$. Substituting the scaled multiplicity bound gives the displayed result; applying the two scaled constraints in Corollary 18 gives its budget-refined version. $\square$

Corollary 20 (Concrete stash security). *For the safe configuration $Z = 64$, $A = 20$, $\mu = 35$, $R = 256$, and $\rho = 0$, the stash-overflow probability is less than 2^{-80} .*

Proof. Set $t = 0.2253$, $q = q_0(t, 35) > 1.4076 > \ln 4$, and $z = e^{-q}$. The fixed-subtree argument underlying Corollary 19, with the budget-refined penalty $\text{dup}_\star$, bounds each root-containing subtree of size n by $e^{20\text{dup}_\star(t) - 256t} z^n$. There are at most $C_n = \frac{1}{n+1} \binom{2n}{n}$ such subtrees; a finite or partially grown tree can only admit fewer. The Catalan generating function gives

$$\sum_{n \geq 1} C_n z^n = \frac{1 - \sqrt{1 - 4z}}{2z} - 1.$$

Thus,

$$\begin{aligned} \Pr[\text{stash} > 256] &\leq e^{20\text{dup}_*(t)-256t} \left(\frac{1 - \sqrt{1 - 4z}}{2z} - 1 \right) \\ &< 2^{-81.6}. \end{aligned}$$

□

We compute the duplicate penalty using this exact expression and choose t to minimize the stash bound. For the complete tree of the fixed-capacity construction in Section 4, Theorem 14 applies with $\mu = \max(A/2, d 2^{-L})$ for d distinct keys present: the first term bounds internal buckets, and the second bounds leaves because an eviction through a leaf merges all copies of every key routing to it. This completes the stash analysis for a fixed tree shape.

C.1 Applying the Bound to Lazy Resizing

Applying the reduction. The second claim of Theorem 13 is exactly the rooted-subtree overflow criterion required by the fixed-shape analysis of Appendix C. The infinite-capacity execution treats physical resizes as no-ops, so its distribution at time t depends on the current logical shape only through P_{Φ_t} . We do not condition on the realized, data-dependent value of Φ_t . Instead, overflow produces one rooted subtree T , and the proof of Theorem 14 already takes a union bound over every rooted binary subtree, including whichever T the realized shape produces. Its tail bound therefore applies without an additional union over resize schedules or shapes.

Derivation of the sweep-load condition. Theorem 13 reduces every instant of a sweep to a fixed shape, a complete tree of height L plus the activated pairs, so the stash bounds of Appendix C apply once their bucket loads are bounded. The required visit pattern is preserved across resizes: doubling the leaf count leaves the low bits of the eviction counter untouched, so every bucket keeps its revisit period and its child alternation, and a frontier bucket enters the shape at its first visit (Lemma 12).

It remains to bound every bucket's load during a sweep. An internal bucket contains at most $A/2$ blocks in expectation by the argument of Appendix C. The argument does not apply to leaves or the frontier, because blocks routed below them are re-collected and re-deposited at every eviction through them. They are bounded by occupancy alone: with d distinct keys present, a node of level L whose children are inactive receives expected load $d 2^{-L}$, and a new leaf half of that. The frontier contribution to the uniform load bound is therefore $d 2^{-L}$ throughout a sweep of the height- L tree.

The resize triggers at occupancy $D 2^L$, and the sweep spans at most $\tilde{A} 2^L$ insertions, during which at most $\tilde{\rho} \tilde{A} 2^L$ fresh keys arrive. The load input of Corollary 19 is therefore $\mu = \max(A/2, D + \tilde{\rho} \tilde{A})$, and $q_\rho(t, \mu) > \ln 4$ is the condition of the lemma. The two bounds cannot be merged into one parameter: the drift integrates the fresh keys of the whole sweep, the tax charges the worst interval between two bucket evictions, and a fresh-heavy prefix followed by a repeat-heavy suffix realizes $\tilde{\rho} = 1$ and $\rho = 0$ simultaneously. Since the refined fixed-subtree decay rate is monotone in each workload parameter at fixed t , checking the corner $\rho = 0, \tilde{\rho} = 1$

covers every stream. The next resize must not fire before the sweep completes, which separately requires $\bar{\rho}\tilde{A} \leq D$. Every quantity scales with 2^L , so the condition is identical at every doubling and the bound holds for the whole run. For distinct-key streams $\rho = \bar{\rho} = 1$, which removes the repeated-key term and, without a shorter sweep interval ($\tilde{A} = A$), leaves the classical Ring ORAM condition at load $Z/2 + A$. Taking $D = Z/2$ recovers the default capacity trigger and load input $\max(A/2, Z/2 + \bar{\rho}\tilde{A})$. The internal-bucket term remains $A/2$, since an interval may begin before the sweep and is then not shortened.

D Proofs for Differentially Oblivious Resizing

In this appendix, we present the proofs of the formal statements made in Section 5.

Theorem 7. *The schedule of resize times produced by Algorithm 3 is $3\epsilon/2$ -DP with respect to neighboring streams (Definition 5).³*

Proof. Fix neighboring streams S, S' that differ on index i and imagine that each are ingested by our algorithm in parallel. Up until i , the execution that takes S has a distribution of resize times that is identical to that of S' . We will argue that the distribution of the timing of the first resize after ingesting the i -th input of S is $\epsilon/2$ -close to that of ingesting the i -th input of S' . Once this is proven, we will make the argument that this price is paid in ≤ 2 later epochs so that basic composition over the 3 epochs yields the claim.

Privacy of First Resize after i . ROGA draws its threshold noise once in `BeginEpoch`, then makes one comparison per noisy measurement interval, and halts at the first positive answer. This is the same pattern that `AboveThreshold` follows; see, e.g., Algorithm 1 in the textbook by Dwork & Roth [27]. Its privacy cost is that of a single noisy measurement regardless of how many comparisons it makes.

One technical difference with the Dwork & Roth text is that we draw the noise for the measurement (η in `ShouldResize`) from the same Laplace distribution as that of the threshold (θ in `BeginEpoch`), while Dwork & Roth use a wider Laplace for the measurement than for the threshold. We choose to use one distribution to simplify presentation as well as ease the proofs of Lemmas 9 & 10. Another point of difference is that Dwork & Roth prove their statements under the assumption that the query sensitivity Δ is 1, while we operate in the $\Delta = 2^L/k$ setting: Δ scales up the count of distinct keys hashed to a leaf and that count has sensitivity 1. To close these gaps, we revise Dwork & Roth’s proof in Appendix E.

Privacy of Subsequent Resizes. Assume $S[i] = a$ while $S'[i] = b$. Let x denote the index of the next instance of a or b , whichever is first. If $S[x] = a$, let y denote the index of the next instance of b otherwise let it denote the index of the next instance of a . Using $\parallel$ to denote concatenation, we can decompose the neighboring input streams into either

$$\begin{aligned} S &= S_1 \parallel S[i] = a \parallel S_2 \parallel S[x] = a \parallel S_3 \parallel S[y] = b \parallel S_4 \\ S' &= S_1 \parallel S[i] = b \parallel S_2 \parallel S[x] = a \parallel S_3 \parallel S[y] = b \parallel S_4 \end{aligned}$$

³We acknowledge the mismatch between ϵ and the proved privacy parameter; a future revision of our work will adjust our pseudocode and implementation to target ϵ -DP.

or

$$\begin{aligned} S &= S_1 \| S[i] = a \| S_2 \| S[x] = b \| S_3 \| S[y] = a \| S_4 \\ S' &= S_1 \| S[i] = b \| S_2 \| S[x] = b \| S_3 \| S[y] = a \| S_4 \end{aligned}$$

Without loss of generality, we will assume the first decomposition holds.

Consider any epoch that does not contain input indices x or y . Its length—the timing until `ShouldResize` returns ‘true’—is independent of whether S or S' was input. The reason is because `ShouldResize` computes $\hat{d}$ from the number of keys that were newly introduced in this epoch, by way of `SortAndMerge`. And by construction, the number of keys that were newly introduced in this epoch does not depend on the input.

Now consider the epochs E_x and E_y that contain index x and y respectively. The expenditure of privacy budget in either epoch is the same $\varepsilon/2$ because sensitivity of $\hat{d}$ is the same. So the worst case is that $\varepsilon/2$ budget is spent on both E_x, E_y on top of the epoch containing index i .

The worst case is realized when $a, b \notin S_1$, meaning that the i -th input was the first time we saw either a or b . The calls to `ShouldResize` in those two epochs will compute different $\hat{d}$ values depending on the input: a is a novel contribution in E_x when S' is the input but not when S is the input, and b is a novel contribution in E_y when S is the input but not when S' is the input.

In the three other cases, at most one of E_x, E_y consumes privacy budget:

1. $a, b \in S_1$. Neither E_x and E_y have data-dependent lengths. This is because a is not a novel contribution in E_x and b is not a novel contribution in E_y : under the same allocation to leaves, both epochs compute the same $\hat{d}$ estimators.
2. $a \in S_1$ but $b \notin S_1$. Here, b is a novel contribution in E_y when S is the input but not when S' is the input.
3. $a \notin S_1$ but $b \in S_1$. Here, a is a novel contribution in E_x when S' is the input but not when S is the input. $\square$

D.1 Resize Timing Guarantees

Lemma 9. *For any epoch, if $r \leq 2^L/k$, resizing occurs before $d_{e,t} > N/2$, except with probability $2(2\alpha)^r$.*

Proof. Let G denote the event that $\theta - \zeta((2\alpha)^r) \leq 0$; by definition of ζ , the failure probability is $(2\alpha)^r$. The rest of the proof will condition on G .

Let t^* be the earliest time for which $d_{e,t^*} \geq N/2 - rAk$. As `ShouldResize` is called every Ak updates, there are at least r calls between t^* and the first time t where $d_{e,t} > N/2$. Writing $\hat{d}[j]$ for the estimator at the j -th such call, Lemma 8 gives $\hat{d}[j] \geq d_{e,t^*} - Ak - \beta(\alpha) \geq N/2 - (1+r)Ak - \beta(\alpha)$ except with probability α . Writing $\eta[j]$ for the accompanying noise sample, we also have $\eta[j] \geq -\zeta(\alpha)$ except with probability α . Thus, the union bound implies that 2α bounds the failure

probability of

$$\begin{aligned}
N/2 + \hat{d}[j] + \eta[j] &\geq N/2 + \hat{d}[j] - \zeta(\alpha) \\
&\geq N/2 + (N/2 - (1+r)Ak - \beta(\alpha)) - \zeta(\alpha) \\
&= \tau - \zeta(\alpha) \\
&\geq \tau - \zeta(\alpha) + (\theta - \zeta((2\alpha)^r)) \\
&= \hat{\tau}.
\end{aligned}$$

In this case, ShouldResize returns ‘true’.

Across the r calls, each $\eta[j]$ is independent. So the intuitive goal is to bound the chance that all calls return ‘false’ by $(2\alpha)^r$. Complicating matters is the fact that the $\hat{d}[j]$ values are not independent. Still, they are derived from mutually distinct sets of leaves, by virtue of how $\{c_i\}_{i \in [k]}$ are constructed and $r \leq 2^L/k$. This correlation is advantageous: when one estimator decreases, the keys arriving in this epoch have a greater probability of mapping to the leaves used by the next estimator, and conversely when it increases.

More precisely, for any positive w , if there are w values of $j' < j$ where $\hat{d}[j'] < N/2 - (1+r)Ak - \beta(\alpha)$, then the probability that $\hat{d}[j] < N/2 - (1+r)Ak - \beta(\alpha)$ conditioned on that event is still bounded by α . Hence the chance that all calls return ‘false’ is bounded by $(2\alpha)^r$. A final union bound with the failure of G yields an overall failure probability of $2(2\alpha)^r$. $\square$

Lemma 10. *If $d_{e,T} \leq N/2 - (1+r)Ak - \beta(\alpha) - \zeta(\alpha) - \zeta((2\alpha)^r) - \zeta(\xi) - \beta(\xi/\lceil T/Ak \rceil) - \zeta(\xi/\lceil T/Ak \rceil)$, then the probability of a resize occurring before time T is $\leq 3\xi$.*

Proof. A resize does not happen at a given check as long as

$$V := N/2 + \hat{d} + \eta - \hat{\tau} \leq 0.$$

It therefore suffices to show that this inequality holds at every check except with probability 3ξ .

There are at most $\lceil T/Ak \rceil$ checks by time T . With all but probability ξ , at every check we have $\eta \leq \zeta(\xi/\lceil T/Ak \rceil)$. Similarly, with all but probability ξ , at every check

$$\hat{d} < d_{e,t} + \beta(\xi/\lceil T/Ak \rceil) \leq d_{e,T} + \beta(\xi/\lceil T/Ak \rceil),$$

where the second inequality uses that $d_{e,t}$ increases with t . Finally, with probability $1 - \xi$, we have $\theta \geq -\zeta(\xi)$. By a union bound, all three properties hold with probability at least $1 - 3\xi$.

Substituting these bounds into V and expanding the definitions of $\hat{\tau}$ and τ gives $V \leq 0$ under the condition in the statement. $\square$

E Modifying AboveThreshold

We distill the resizing logic in ROGA into the pseudocode of Algorithm 4. We borrow notation from the AboveThreshold pseudocode of Dwork & Roth. The two differences with the prior work is the use of $\Delta \geq 1$ sensitivity and the use of exactly one Laplace distribution in both sources.

We can obtain the desired result for the proof of Theorem 7—the distribution of the resize time for any single epoch is $\varepsilon/2$ -private—by defining f_i to be the computation of $\hat{d}$ in the i -th call to `ShouldResize`, defining T to be $\tau - 2\zeta$, and replacing each instance of ε with $(\varepsilon/3)$.

```

let  $\hat{T} \leftarrow T + \text{Lap}(2\Delta/\varepsilon)$ 
for  $i \in \mathbb{N}$ 
  let  $v_i \leftarrow \text{Lap}(2\Delta/\varepsilon)$ 
  if  $f_i(S) + v_i \geq \hat{T}$  :
    output  $a_i \leftarrow \top$  (shorthand for True)
    halt
  else
    output  $a_i \leftarrow \perp$  (shorthand for False)

```

Algorithm 4: `ModAboveThreshold`, `AboveThreshold` modified to use one noise distribution and parameterized by sensitivity Δ . It takes a stream S , Δ -sensitive queries $\{f_i\}$, threshold T , and privacy budget ε , and outputs a stream $a_1, \dots$.

Theorem 21. For functions $\{f_i\}$ that are Δ -sensitive, `ModAboveThreshold` satisfies $3\varepsilon/2$ -DP.

Proof. For any integer $k \geq 1$, we will argue

$$\begin{aligned} & \mathbb{P}[\text{ModAboveThreshold}(S) = \perp^{k-1} \top] \\ & \approx_\varepsilon \mathbb{P}[\text{ModAboveThreshold}(S') = \perp^{k-1} \top] \end{aligned}$$

only over the randomness of $\hat{T}$ and v_k : fixing $v_1, \dots, v_{k-1}$, define the quantity

$$g(S) := \max_{i < k} f_i(S) + v_i$$

so that

$$\begin{aligned} & \mathbb{P}[\text{ModAboveThreshold}(S) = \perp^{k-1} \top] \\ &= \mathbb{P}[\hat{T} > g(S) \text{ and } f_k(S) + v_k \geq \hat{T}] \\ &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mathbb{1}[t \in (g(S), f_k(S) + v)] \\ & \quad \cdot \mathbb{P}[\hat{T} = t] \mathbb{P}[v_k = v] dv dt \end{aligned}$$

where $\mathbb{P}[\hat{T} = t], \mathbb{P}[v_k = v]$ denote density of the probability distributions evaluated at t, v respectively.

We now define the following

$$\begin{aligned} \hat{v} &:= v + g(S) - g(S') + f_k(S') - f_k(S) \\ \hat{t} &:= t + g(S) - g(S') \end{aligned}$$

Observe that our sensitivity bound directly implies $|\hat{t} - t| = |g(S) - g(S')| \leq \Delta$. We also have $|\hat{v} - v| = |g(S) - g(S') + f_k(S') - f_k(S)| \leq |g(S) - g(S')| + |f_k(S') - f_k(S)| \leq 2\Delta$

We perform a change of variables:

$$\begin{aligned}
& \mathbb{P}[\text{ModAboveThreshold}(S) = \perp^{k-1} \top] \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mathbb{P}[\hat{T} = \hat{t}] \cdot \mathbb{P}[v_k = \hat{v}] \\
&\quad \cdot \mathbb{1}[\hat{t} \in (g(S), f_k(S) + \hat{v})] \, dv \, dt \\
&= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} \mathbb{P}[\hat{T} = \hat{t}] \cdot \mathbb{P}[v_k = \hat{v}] \\
&\quad \cdot \mathbb{1}[t \in (g(S'), f_k(S') + v)] \, dv \, dt \\
&\leq \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} e^{\varepsilon/2} \mathbb{P}[\hat{T} = t] \cdot e^{\varepsilon} \mathbb{P}[v_k = v] \\
&\quad \cdot \mathbb{1}[t \in (g(S'), f_k(S') + v)] \, dv \, dt \\
&= e^{3\varepsilon/2} \cdot \mathbb{P}[\hat{T} > g(S') \text{ and } f_k(S') + v_k \geq \hat{T}] \\
&= e^{3\varepsilon/2} \cdot \mathbb{P}[\text{ModAboveThreshold}(S') = \perp^{k-1} \top].
\end{aligned}$$

The inequality comes from our choice of Laplace parameter and our bounds $|\hat{t} - t| \leq \Delta$, $|\hat{v} - v| \leq 2\Delta$. $\square$

Remark 22. Prior work by Bell-Clark et al. show that a tighter bound of ε -DP is achievable if the queries $\{f_i\}$ also satisfy unidirectional sensitivity [11]. In our case, the $\hat{d}$ estimators in `ShouldResize` are derived from disjoint sets of indicators, which precludes that property.