Memory Checking Requires Logarithmic Overhead

Elette Boyle[*] Ilan Komargodski[†] Neekon Vafa[‡]

Abstract

We study the complexity of memory checkers with computational security and prove the first general tight lower bound.

Memory checkers, first introduced over 30 years ago by Blum, Evans, Gemmel, Kannan, and Naor (FOCS '91, Algorithmica '94), allow a user to store and maintain a large memory on a remote and unreliable server by using small trusted local storage. The user can issue instructions to the server and after every instruction, obtain either the correct value or a failure (but not an incorrect answer) with high probability. The main complexity measure of interest is the size of the local storage and the number of queries the memory checker makes upon every logical instruction. The most efficient known construction has query complexity $O(\log n/\log\log n)$ and local space proportional to a computational security parameter, assuming one-way functions, where n is the logical memory size. Dwork, Naor, Rothblum, and Vaikuntanathan (TCC '09) showed that for a restricted class of "deterministic and non-adaptive" memory checkers, this construction is optimal, up to constant factors. However, going beyond the small class of deterministic and non-adaptive constructions has remained a major open problem.

In this work, we fully resolve the complexity of memory checkers by showing that *any* construction with local space p and query complexity q must satisfy

$$p \geq \frac{n}{(\log n)^{O(q)}} \ .$$

This implies, as a special case, that $q \geq \Omega(\log n/\log\log n)$ in any scheme, assuming that $p \leq n^{1-\varepsilon}$ for $\varepsilon > 0$. The bound applies to any scheme with computational security, completeness $2/3$, and inverse polynomial in n soundness (all of which make our lower bound only stronger). We further extend the lower bound to schemes where the read complexity q_r and write complexity q_w differ. For instance, we show the tight bound that if $q_r = O(1)$ and $p \leq n^{1-\varepsilon}$ for $\varepsilon > 0$, then $q_w \geq n^{\Omega(1)}$. This is the first lower bound, for any non-trivial class of constructions, showing a read-write query complexity trade-off.

Our proof is via a delicate compression argument showing that a "too good to be true" memory checker can be used to compress random bits of information. We draw inspiration from tools recently developed for lower bounds for relaxed locally decodable codes. However, our proof itself significantly departs from these works, necessitated by the differences between settings.

[*] Reichman University and NTT Research. Email: eboyle@alum.mit.edu.

[†] Hebrew University and NTT Research. Email: ilank@cs.huji.ac.il.

[‡] MIT. Email: nvafa@mit.edu.

1 Introduction

Consider a user who wishes to maintain and operate on a large database but does not have sufficient local memory. A natural idea is for the user to offload the database to a remote storage service and perform accesses remotely. This solution, however, introduces a trust concern: the user must trust the storage service to perform its accesses reliably. Blum, Evans, Gemmel, Kannan and Naor [BEG$^+$94] addressed this issue more than 30 years ago by introducing the concept of a *memory checker*: a method for a user to use its small (but trusted) local storage to detect faults in the large untrusted storage service. Since cloud storage and cloud computing have become widespread and growing practices, the need to guarantee the integrity of remotely stored data is paramount. Beyond merely checking integrity of remotely stored data [BEG$^+$94, HJ05, CSG$^+$05, OR07, CD16], memory checkers have found applications in various other real-world applications, for example, provable data possession and retrievability systems [ABC$^+$07, JJ07, OR07, SW13, CKW17], various verifiable computation systems [WTS$^+$18, XZZ$^+$19, Set20, ZXZS20, OWWB20, BMM$^+$21, BDFG21, BCHO22], and many more.

A memory checker can be thought of as a proxy between the user and the untrusted remote storage. The checker receives from the user an adaptively generated sequence of read and write operations to a large unreliable memory. For each such logical request, it makes its own physical queries to the remote storage. The checker then uses the responses, together with a small reliable local memory, to either ascertain the correct answer to the logical request or report that the remote storage was faulty. The checker's assertions should be correct with high probability; typically, a small two-sided error is permitted. The main complexity measures of a memory checker are its **space complexity** (denoted p), the size of the reliable local memory in bits, and its **query complexity** (denoted q), the number of physical queries made to the unreliable memory per logical user request.

Blum et al.'s main results are efficient memory checkers in two different settings, as stated next. Throughout, we use n to denote the (logical) memory size.

- <u>Construction 1</u>: Space complexity $p = O(\lambda)$, where λ is the size of a cryptographic key, and query complexity $q = O(\log n)$. These constructions assume that one-way functions exist and that the adversary who controls the unreliable memory is computationally efficient.

- <u>Construction 2</u>: Space complexity p and query complexity $q = O(n/p)$. This construction is statistically secure.

Naor and Rothblum [NR09] showed that any memory checker with a non-trivial query-space tradeoff (i.e., $q = o(n/p)$) must be computationally secure and must be based on the existence of one-way functions. Thus, Construction 2 from above is optimal among all statistically secure constructions. Yet, whether more efficient computationally secure constructions than Construction 1 exist is a long-standing open problem. In particular, is the logarithmic query complexity necessary? In many applications, logarithmic overhead per query is a significant price to pay for data verification, so if more efficient constructions exist, they may be preferable.

While this problem has been open for more than three decades, only one work managed to make progress. This work, due to Dwork, Naor, Rothblum, and Vaikuntanathan [DNRV09], showed that logarithmic query complexity is inherent for a *restricted class of memory checkers*. They consider memory checkers where for each read/write operation made by the user, the locations that the checker accesses in the unreliable memory are fixed and known. They refer to such checkers as *deterministic and non-adaptive*. While this class captures many known memory checker constructions, it is obviously quite restrictive. For instance, a memory checker could be *non-deterministic*, i.e., decide on its queries to the unreliable memory in a probabilistic manner via randomness derived from freshly sampled coins. Alternatively, a memory checker could be *adaptive*, i.e., choose sequentially which locations in the remote storage it reads and writes, and choose these locations based on the contents of earlier read locations or its reliable local memory contents. Of course, a memory checker could be *both* non-deterministic and adaptive, potentially resulting with more efficient construction.

Indeed, there are many computational models where randomness and/or adaptivity are either necessary or make certain problems easier. For instance, in the context of two-party communication complexity it is well known (see [NW93]) that for every $k \in \mathbb{N}$, randomization is more powerful than determinism in k-round protocols, and further that having k rounds of adaptive communication is (exponentially) better than having only $k-1$ rounds. This shows that randomness and adaptivity are computational resources that have the potential to significantly improve the complexity of certain tasks. Additionally, in the context of *oblivious* RAM computation (which we shall discuss in length below), randomness is known to be necessary [GO96] and there is evidence that adaptivity is required in certain non-trivial regimes of parameters [CDH20].

Thus, the main question we address in this work is as follows:

What is the achievable complexity of memory checkers?
Do memory checkers with sub-logarithmic query complexity exist (under any cryptographic assumption)?

1.1 Our Results

We fully resolve the complexity of memory checkers by showing that logarithmic query complexity is inherent, no matter how the scheme operates and no matter which computational assumptions are used. Specifically, we prove the following theorem.

Theorem 1 (Informal; see Theorem 5)**.** *Every memory checker (with computational security) for a logical memory of size n that has query complexity q, and local state p, must satisfy $p \geq \frac{n}{(\log n)^{O(q)}}$.*

In particular, $q \geq \Omega(\log n / \log\log n)$ assuming that $p \leq n^{1-\varepsilon}$ for $\varepsilon > 0$. As mentioned, the above theorem applies to all possible schemes, including ones that use randomness and adaptivity to decide which locations to access in the remote storage, and also in the computational setting. Furthermore, the lower bound applies even to schemes where the local state is private (from the adversary). Lastly, the completeness of the memory checker in the theorem is $2/3$ and soundness is inverse polynomial in n. Note that these make our result only stronger because in constructions we usually aim for perfect (or near-perfect) completeness and negligible soundness.

The lower bound is tight up to the constant hidden in the $O(\cdot)$ notation. A memory checker construction with matching complexity, i.e., with $q = O(\log n / \log\log n)$, was given by Papamanthou and Tamassi [PT11]. However, their construction requires *private* local state (used to store a secret PRF key). We improve upon their result and show (for completeness) a construction with quantitatively matching complexity and with only (public) *reliable* local state. Our construction requires the existence of sub-exponentially secure one-way functions, as is needed in all other computationally secure memory checker constructions with similar properties.

Theorem 2 (Informal; see Corollary 5)**.** *Assume the existence of sub-exponentially secure one-way functions. For all sufficiently large $q \leq \log n / \log\log n$, there is a deterministic and non-adaptive* *memory checker for a logical memory of size n with* reliable *local state, perfect completeness, and (computational) negligible soundness in n, that has local space $p \leq \frac{n}{(\log n)^{\Omega(q)}}$.*

In particular, for some $q = \Theta(\log n / \log\log n)$, this construction has local space $p = \mathsf{polylog}(n)$.

We refer to Figure 1.1 for a summary of known memory checker constructions.

Reference	Read complexity	Write complexity	Secret state	Remark
[BEG$^+$94]	$O(\log n)$	$O(\log n)$	No	
[DNRV09]	$O(d \log_d n)$	$O(\log_d n)$	Yes	d is arbitrary
[DNRV09]	$O(\log_d n)$	$O(d \log_d n)$	Yes	d is arbitrary
[PT11]	$O(\log n / \log\log n)$	$O(\log n / \log\log n)$	Yes	
This work	$O(\log n / \log\log n)$	$O(\log n / \log\log n)$	No	Corollary 4

Figure 1: Complexity of memory checker constructions for a logical memory of size n.

Reads vs. writes. Our lower bound rules out memory checkers where the worst-case query complexity of all accesses is below (quasi-)logarithmic. However, not all types of accesses necessarily occur as often in applications. Depending on the application, it could be that read operations are far more frequent than writes, or vice versa. Our stated lower bound from above does not rule out a memory checker where the read complexity is, say, constant but writes have logarithmic complexity. In fact, no known lower bound, not even for restricted classes of constructions (e.g., deterministic and non-adaptive), rules out such a construction. We extend our lower bound from Theorem 1 to this setting and prove a general trade-off between the local reliable space, the read query complexity, and the write query complexity.

Theorem 3 (Informal; see Theorem 5)**.** *Every memory checker (with computational security) for a logical memory of size n that has read-query complexity q_r, write-query complexity q_w, and local state p, must satisfy $p \geq \frac{n}{(q_r q_w \log n)^{O(q_r)}}$.*

In particular, the above theorem rules out a memory checker with constant read-query complexity and sub-polynomial write-query complexity, i.e., every memory checker with $q_r = O(1)$ and local space $p \leq n^{1-\Omega(1)}$ must have $q_w = n^{\Omega(1)}$. This bound is optimal due to a construction of Dwork et al. [DNRV09] who showed a construction with write-query (resp. read-query) complexity $O(d \log_d n)$ and read-query (resp. write-query) complexity $O(\log_d n)$ for every parameter d. Indeed, setting $d = n^{O(1)}$, we get a memory checker with $q_r = O(1)$ and $q_w = n^{O(1)}$, which is optimal according to our lower bound. Finally, we note that Theorem 1 is obtained as a special case of Theorem 3 with $q_r = q_w$.

Interestingly, we do not know if the "reverse" bound also holds in general, i.e., that if $q_w = O(1)$ then q_w must be large. We leave this as an intriguing open problem. We make initial steps towards this question by proving it for the restricted class of deterministic and non-adaptive memory checkers.

Theorem 4 (Informal; see Theorem 6)**.** *Every deterministic and non-adaptive memory checker (with computational security) for a logical memory of size n that has read-query complexity q_r, write-query complexity q_w, and local state p, must satisfy $p \geq \frac{n}{(q_r q_w \log n)^{O(\min\{q_w, q_r\})}}$.*

This theorem means that if either one of q_r or q_w are sub-(quasi)-logarithmic, then the other one must be much larger. In other words, the "reverse" bound of Theorem 3 holds for deterministic and non-adaptive constructions: if $q_w = O(1)$, then necessarily $q_r = n^{\Omega(1)}$ (as long as the local space is not too large, i.e., $p \leq n^{1-\Omega(1)}$). By the above-mentioned constructions of Dwork et al. [DNRV09] (that happen to be deterministic and non-adaptive), we conclude that our lower bound is tight, fully resolving the complexity of deterministic and non-adaptive memory checkers. Lastly, we mention that we provide some (weak) evidence that a "reverse" bound of Theorem 3 for general (not necessarily deterministic and non-adaptive) schemes would require relatively new ideas; see Appendix C for details.

We refer to Figure 1.1 for a summary of known memory checker lower bounds.

Reference	**Space/Query tradeoff**	**Limitation of the Scheme**
[DNRV09]	$p \geq n/(\log n)^{O(\max\{q_r,q_w\})}$	Deterministic and non-adaptive
This work	$p \geq n/(\log n)^{O(q_r)}$	None
	$p \geq n/(\log n)^{O(\min\{q_r,q_w\})}$	Deterministic and non-adaptive

Figure 2: Lower bounds on the local space p of memory checkers for a logical memory of size n with q_r read query complexity and q_w write query complexity.

1.2 Implications of our Lower Bounds

Lower bounds for memory checkers optimizing other metrics. As an immediate application of our lower bound, we get general lower bounds for memory checking in other models where different notions of efficiency are considered. We mention two recent works next. Mathialagan [Mat23] extended the memory checking notion to deal with Parallel RAM (PRAM) machines, and suggested a construction for PRAMs with m CPUs with $O(\log N)$ query blowup and $O(\log N)$ depth blowup, relying on the existence of one-way functions. Since their constructions match (in query complexity) the best known construction in the RAM setting, along with our lower bound, we conclude that their scheme is optimal in this sense. Wang, Lu, Papamanthou, and Zhang [WLPZ23] studied the locality of memory checkers (i.e., the number of non-contiguous memory regions a checker must query to verifiably answer a read or a write query). They adapted the lower bound of Dwork et al. [DNRV09] to conclude that $\Omega(\log n/\log\log n)$ locality is necessary for any deterministic and non-adaptive memory checker. Our lower bound implies (analogously) that the same lower bound applies to all possible schemes.

Impossibility of efficient black-box malicious Oblivious RAM compilers. As mentioned, memory checkers are used to solve the trust issue that arises when one offloads a memory to a remote and untrusted server. However, there is also a privacy concern in doing so which is not addressed by memory checkers. Since the remote server fully controls the memory and executes instructions in the user's behalf, then it can see the user's data and the program being executed. To obtain privacy, we need to hide the data (using say an encryption scheme) and also "scramble" the observed access patterns so that instructions look unrelated to the data or the program being executed. The tool that achieves the latter goal is called *Oblivious RAM* (ORAM), introduced in the seminal works of Goldreich and Ostrovsky [Gol87, Ost90, GO96].

The efficiency of ORAM schemes is measured (similarly to memory checkers) by the number of physical queries in the oblivious simulation per logical database query. We know that logarithmic overhead is unavoidable, for any construction, even ones that rely on cryptographic assumptions [GO96, BN16, LN18, KL21]. We also have matching constructions with (worst-case) logarithmic overhead, where security is computational and relying on one-way functions [PPRY18, AKL+23, AKLS21].

We note however that in the ORAM setting, one typically assumes that the remote untrusted server is passive in the sense that it behaves honestly except that it tries to learn information about the underlying data or program from the observed access pattern. A natural question is whether it is possible to tolerate a malicious attacker that may not behave honestly and actually tamper with the memory while trying to learn something about the underlying data. It is well-known (e.g., [LPM+13]) that by naively compiling every instruction of the ORAM using a memory checker one obtains a "maliciously secure" ORAM that achieves both privacy and integrity, simultaneously. This however comes at a cost: every logical instruction will now require $\textsc{ORAMOverhead}\times\textsc{MemoryCheckerOverhead}$ physical accesses, which is roughly $\Theta(\log^2 n)$ (using [BEG+94, AKL+23]).[1]

Mathialagan and Vafa [MV23] recently improved upon the above generic compiler and gave a construction (called MacORAMa) of a"maliciously secure" ORAM with overhead $O(\log n)$ (which is obviously optimal). They achieve their result by a tedious process of opening up every building block in [AKL+23, AKLS21]'s construction and turning it into a maliciously secure building block using various constructions of memory checkers. As such, the construction is overall very long and complex. [MV23] ask if such a white-box construction and analysis is necessary or maybe there is a generic way to compile ORAMs into maliciously secure counterparts with only constant overhead. Towards this, a barrier was presented by [MV23]: such a "black-box" compiler would imply a memory checker with $O(1)$ query complexity. By our result (Theorem 1), such memory checkers do not exist, no matter what, and so there is no way to generically upgrade ORAMs into malicious with less than additional logarithmic blowup. See details and the formal statement in Appendix A.

It is crucial for this corollary that our lower bound in Theorem 1 applies to all constructions of memory checkers, including ones that use randomness and adaptivity. Indeed, the way Mathialagan and Vafa obtain the above-mentioned "barrier" is by using an ORAM and the malicious compiler to build a memory checker. Since ORAMs are inherently randomized and (as far as we know [CDH20]) require adaptivity, their resulting memory checker is using randomness and adaptivity.

1.3 Offline vs. Online Memory Checkers

We have considered memory checkers that need to report either a correct value or an error (but never be wrong) after every logical instruction. This notion of memory checkers is known as "online" memory checkers. Prior works in this area additionally considered a weaker notion called "offline" memory checkers. The latter are required to report that some error has occurred only after a batch of requests. None of our lower bounds apply to this weaker notion, and this is not surprising: there exist offline memory checkers that achieve amortized $O(1)$ bandwidth (see [BEG+94] and [DNRV09, Section 5]); these are even statistically secure and do not require any cryptographic assumptions.

[1] For specific ORAM constructions such as Path ORAM there are more efficient composition methods, e.g., [SvDS+13, RFY+13], but since the ORAM itself is sub-optimal, they result in similar $\approx \log^2 n$ overall complexity.

2 Technical Overview

We begin by describing the previous approach of Dwork et al. [DNRV09], which as we mentioned, only gives a lower bound for *deterministic and non-adaptive* memory checkers. We already mention that our approach significantly differs from theirs, and the main purpose of this part of the overview is to explain why their approach seems only applicable to the restricted class of deterministic and non-adaptive constructions.

At a high level, they proceed as follows. First, they prove a lower bound for a base case setting where the query complexity of the memory checker is 1; this is done by a compression argument. Then, they consider the general case, and show a reduction from high query complexity to lower complexity at the expense of decreasing the logical memory size, increasing the local space, and increasing the physical word size. The reduction is performed iteratively until they end up with a memory checker with query complexity 1, where they can invoke the base case lower bound. More details follow.

Base case. Suppose there is a deterministic and non-adaptive memory checker with query complexity 1: namely, a single *fixed* physical query takes place for each logical query. To obtain a lower bound for this base case, they invoke a compression argument, where Alice is to transmit a random string $x \sim \{0,1\}^n$ to Bob. Consider a sequence of logical operations that first writes 0 to each logical index, resulting in public database DB_0 and local state st_0. (Alice and Bob can share DB_0 and st_0 for free since there is no dependence on x.) Then, Alice uses the memory checker to write 1 to all $i \in [n]$ such that $x_i = 1$, resulting in public database DB_1 and local state st_1. Alice will send just st_1 to Bob.

Bob's decoding strategy is as follows: for all $i \in [n]$, use the memory checker to read logical index i using DB_0 and st_1. If the answer is a bit value $b \in \{0,1\}$ value, set $\tilde{x}_i = b$. Otherwise, if the answer is $\perp$, set $\tilde{x}_i = 1$. By soundness, for all i such that the answer is a bit value, we know $\tilde{x}_i = x_i$. By completeness,[2] since queries to logical indices i must induce disjoint physical queries (here relying on query complexity 1 and basic information encoding), if $x_i = 0$, then we must recover the bit value 0, since DB_0 and DB_1 are consistent at the corresponding ith physical query location. Therefore, for all $i \in [n]$, $\tilde{x}_i = x_i$. Since Alice and Bob communicated n bits of information, it follows by a counting argument that $|\mathsf{st}_1| \geq n$.

Reducing to the base case. Next, they show to reduce the query complexity of a general memory checker to 1, at the cost of degrading the other parameters (the logical memory size, the local space, and the physical word size). Roughly speaking, they condition on the following: either there is a set of sufficiently many physical locations that are sufficiently "heavy," in the sense that many logical queries touch that location, or there does not exist such a set. In either case, the assumption that the construction is deterministic and non-adaptive is crucial.

- If there does exist such a heavy set, then, they restrict the memory checker to logical indices that frequently touch that heavy set. In this case, they move the heavy part of the public database into the local state and reduce query complexity, at the cost of decreasing the logical memory size and increasing the local space.

[2] They assume throughout that logical reads and writes query the same physical locations, at the cost of a multiplicative factor of 2 blowup in query complexity in the ultimate scheme.

- If there does *not* exist such a heavy set, they argue that one can prune off (not too many) logical indices so that the remaining logical indices have disjoint physical locations. By restricting the memory checker to these logical indices and increasing the physical word size, this becomes a memory checker with query complexity 1. (Then, the base case applies.)

Without significant new ideas, it seems hard to use the above template to go beyond deterministic and non-adaptive constructions, as we argue next. Both of the above main steps, i.e., the base case and the reduction to the base case, rely on this assumption. We would thus need to prove an analogue of the base case for general constructions and then devise a reduction that works even if the construction uses randomness and adaptivity. We believe that the harder task is the latter one. It is not at all clear how one could restrict logical indices in either of the two cases. In the first case, one would need to at least define the heavy set differently, and in the second case, because of randomness and adaptivity, it is possible that a memory checker can adaptively choose to have many logical queries overlapping in random locations, making no single location "heavy" and yet there will be no non-trivial subset of logical indices with disjoint physical locations.

2.1 Our Lower Bound

Like Dwork et al. [DNRV09], we will also use a compression argument using the local state of the memory checker, but this is where the resemblance of our proofs ends. Their proof restricts memory checkers into smaller and smaller ones, finally arriving at a base case compression argument. Ours, on the other hand, will directly run a compression argument in "one shot," without reducing the memory checker into smaller ones.

Below, let the memory checker use physical words (i.e., blocks) of size $w \leq \mathsf{polylog}(n)$ bits. Furthermore, we let m denote the size (i.e., number of words) of the remote server.

Suppose Alice wishes to communicate a random string of $x \in \{0,1\}^n$ of Hamming weight k to Bob (where k is some parameter set later). Alice and Bob can first initialize a memory checker with "0"s logically written everywhere, producing some public database $DB_0 \in (\{0,1\}^w)^m$, independent of x. Then, Alice can use this memory checker to write the value "1" to each of these indices $i \in [n]$ where $x_i = 1$, which will result in some new, updated public database, $DB_1 \in (\{0,1\}^w)^m$. After doing this, Alice can send to Bob the resulting local state st_1 of the memory checker. Note, however, that Bob does not see the updated database entries DB_1.

Bob can then use the following decoding strategy to extract x: using st_1 from Alice and the (outdated) public database DB_0 from initialization, sequentially emulate a logical read operation on the memory checker for each $i \in [n]$, rewinding the local state st_1 back between each logical read. Each such query is equivalent to a true operation of the memory checker, with an adversarial remote memory that replaces the true DB_1 values with outdated ones from DB_0. From the syntax of the memory checker, for each i, Bob will receive either 0, 1, or $\perp$. By the *soundness* guarantee of the memory checker, if Bob receives 0 or 1 for some i, Bob knows it is the correct value of x_i (with high probability). However, Bob cannot conclude anything if receiving $\perp$ from the memory checker. Our hope will be to leverage the *completeness* guarantee of memory checker, which says that if the memory checker always sees correct (i.e., fresh) values of the public database—namely, DB_0 is equal to DB_1 in all queried locations—then Bob will derive the correct binary, non-$\perp$ value. The challenge remains of how to do so, given that DB_1 in fact could differ largely from DB_0.

More precisely, let $W \subseteq [m]$ represent the set of physical locations that were written to in Alice's k writes (so DB_0 and DB_1 differ only on W). In particular, $|W| \leq kq$, where q is the query complexity to remote memory for each logical request. If for a given read query, the physical locations accessed in $[m]$ indeed avoid W, then we would be done, since Bob's decoding strategy can recover x by invoking completeness (and soundness) of the memory checker. However, there is no reason that this guarantee should be true. For example, in Merkle-tree style constructions, the root of the Merkle tree is accessed for *all* logical queries.

One possible way to get around this is to partition the public database into "heavy" and "light" locations. For heavy locations (e.g., the root of a Merkle tree), one can add their contents to the local space (or equivalently, in the communication game, have Alice send the contents to Bob), and for the light locations (e.g., lower levels of a Merkle tree), we hope that W does not hit too many of them. It turns out, however, that such a naive approach will not give us a useful lower bound, as we explain a bit later (see Remark 1), after we develop some of our ideas further. Thus, we will need a more intricate query partitioning mechanism that we explain next.

Tri-partitioning the public database. We use a more fine-grained partition into *heavy*, *medium*, and *light* locations $z \in [m]$, denoted by the sets $H, M, L \subseteq [m]$, respectively, in the following sense: when taking a uniformly random logical index $i \sim [n]$ and reading it with local state st_1 and public database DB_1, and sampling a uniformly random one of the corresponding q physical queries, what is the probability that it equals z? For thresholds implicitly set later, H will contain the physical locations z with highest probability, L the lowest probability, and M everything in between. Importantly, the fact that we make the heavy and light sets further apart by introducing the middle set M is crucial for us; see Remark 1.

With this partition in hand, we adjust our communication protocol between Alice and Bob as follows. As mentioned above, we send the heavy locations in the clear; i.e., after Alice performs the k writes, Alice will additionally send over $DB_1|_H$ (instead of just st_1), i.e., the updated contents of the physical locations in the heavy set H.[3]

Now, Bob has access to some "hybrid" public database $\widetilde{DB}$, defined as DB_1 on H and DB_0 on $L \cup M$, and Bob will try reading from $\widetilde{DB}$ instead of DB_0 (still using the updated local state st_1). The set of "bad" physical locations that we are worried about is $\mathrm{BAD} := W \cap (M \cup L) \subseteq [m]$. As long as any given read from Bob avoids BAD in all of its q physical queries, we can invoke completeness to say that Bob will receive the correct answer for that logical index.

We can decompose BAD into $(W \cap M) \cup (W \cap L)$ and analyze both cases separately. First, the medium thresholds for M will have the property that for random $i \sim [n]$, the probability that *all* q physical queries for logical read i avoid M is at least (say) $99/100$. Second, the light threshold for L will have the property that for all $\ell \in L$, the probability (over random $i \sim [n]$ and a random one of the q physical queries) that the location is ℓ is at most δ. By a union bound, this means that a random read to $i \sim [n]$ will hit ℓ for *at least one* of the q physical queries with probability at most $q\delta$. Since $W \cap L \subseteq L$ and $|W \cap L| \leq kq$, the probability that a random logical read to $i \sim [n]$ hits $W \cap L$ is at most $k\delta q^2$. We now set $k = \Theta(1/(\delta q^2))$ so that this probability is at most (say) $1/100$. Therefore, in total, the probability that a random read to $i \sim [n]$ avoids BAD is at least $98/100$. This implies that Bob will be able to recover at least $\Omega(k)$ bits of information about the random string x. Thus, by sending st_1 and $DB_1|_H$, Alice has communicated $\Omega(k)$ bits about x.

Let $p = |\mathsf{st}_1|$ denote the local space of the memory checker. By a standard encoding lemma, saying that the most communication efficient way to transmit uniformly random ℓ bits of information

[3] More precisely, for $H \subseteq [m]$ and $DB_1 \in (\{0,1\}^w)^m$, we define $DB_1|_H \in (\{0,1\}^w)^{|H|}$ to be the restriction of DB_1 to indices in H.

is by sending them in the clear, we get the inequality

$$p + |H|w \geq \Omega(k).$$

Rearranging and using our setting of $k = \Theta(1/(\delta q^2))$, this means

$$p \geq \frac{1}{\Theta(\delta q^2)} - |H|w. \tag{1}$$

Remark 1 (On the necessity of the medium set)**.** *We now explain why we need a "medium" set, M. If we had just a heavy and a light set, the same analysis as above applies. However, it is possible that $|H|w$ and $1/\Theta(\delta q^2)$ do not have enough of a gap, in which case the right-hand side of* (1) *becomes very small. The medium set allows for this gap to be large.*

To explicitly see why this approach fails if there is no medium set, consider the following setup. Suppose $w = 4\ln(m)$ and we have a distribution[4] *over $[m]$ elements as follows:*

$$\Pr[i] := \begin{cases} \frac{1}{2} & i = 1, \\ \frac{1 \pm o(1)}{2\ln(m)(i-1)} & i \geq 2. \end{cases}$$

For all choices of L and δ, as long as $L \neq \emptyset$, we have $1/\delta - |H|w \leq 2$, which is tight when $L = [m]$ and $\delta = 1/2$. (Note that if $L = \emptyset$, then Alice is sending the whole updated database to Bob, which has no compression.) This renders (1) *useless.*

All that is left is the following: how exactly do we set our thresholds for H, M, L (and thus δ and $|H|$) to maximize the right hand side, where the probability that all q physical queries avoid M is at least 99/100? To do this, we prove a generic partition lemma (whose proof can be found in Section 3):

Lemma 1 (Partition Lemma (informal); see Lemma 3)**.** *Let X be a random variable supported on a finite set S. Let $\gamma > 1$, and let $c, n \in \mathbb{N}$. Then, there is a partition $S = L \sqcup M \sqcup H$ such that the following hold for some $\delta \geq 1/n$:*

- *$\Pr[X = \ell] \leq \delta$ for all $\ell \in L$,*
- *$\Pr[X \in M] \leq 1/c$, and*
- *The set H satisfies*
$$\frac{1}{\delta} - |H|\gamma > \frac{n}{(2\gamma)^c}.$$

Now, we can directly use Lemma 3 with $S = [m]$, $\gamma = \Theta(q^2 w)$, and $c = 100q$ on the distribution over $[m]$ described earlier. Plugging this back into (1), we immediately have

$$p > \frac{n}{(q^2 w)^{O(q)}}.$$

In particular, if $w \leq \mathsf{polylog}(n)$ and $p \leq n^{1-\epsilon}$ for some $\epsilon > 0$, then $q = \Omega(\log n / \log\log n)$.

[4] This distribution comes from Goldreich [Gol23].

Read and write query complexities. We can modify the above analysis to the setting where the read query complexity (q_r) and the write query complexity (q_r) differ. Then, we set $c = 100q_r$ and $k = \Theta(1/\delta q_r q_w)$, which arrives at

$$p > \frac{n}{(q_r q_w w)^{O(q_r)}}.$$

Therefore, for example, if $q_r = O(1)$, $p \leq n^{1-\epsilon}$ for some $\epsilon > 0$, and $w \leq \mathsf{polylog}(n)$, then $q_w = n^{\Omega(1)}$. See Corollary 2 for more details.

Loose ends. There are a couple of issues swept under the rug in the above exposition.

- First, the distribution over physical indices produced by the logical read can adaptively change after each write, so Bob does not know what H, M, or L are.
- Second, our setting of parameters is circular. That is, we set $k = \Theta(1/(\delta q^2))$, and k is part of the description of the communication game that Alice and Bob are playing. However, δ is only given after applying the partition lemma (Lemma 3), which itself depends adaptively on the behavior of the memory checker.

For the first issue, we observe that Bob only needs to know what H is, so that Bob can produce $\widetilde{DB}$. Specifying $H \subseteq [m]$ can be done with $|H| \cdot \lceil \log_2 m \rceil$ bits, so Alice can send this as well in her message to Bob for free as long as $w \geq \Omega(\log n)$ and $m \leq \mathsf{poly}(n)$.[5]

For the second (and more challenging) issue, we modify the communication game as follows. Alice and Bob together effectively "guess" a good value of δ before beginning the protocol. In doing so, Alice can also always feed the memory checker the same number of writes (regardless of δ), and will instead set the "initialized" database DB_0 to be after some number of writes depending on δ. Thankfully, our proof of the partition lemma additionally guarantees that there are most $c = \Theta(q)$ possible choices for δ, so guessing δ correctly occurs with noticeable $\Theta(1/q)$ probability.

Connection to Relaxed Locally Decodable Codes. Our technique for partitioning $[m]$ into heavy, medium, and light queries is inspired from a work of Goldreich [Gol23] in revisiting lower bounds on the length of relaxed locally decodable codes (rLDCs).[6] In their setting, partitioning indices of the codeword into heavy, medium, and light is done in a different manner and for different reasons. For example, instead of having Alice send Bob the heavy set (as is done above), in the rLDC setting, each logical index $i \in [n]$ has a *different* corresponding heavy set H_i, so Alice cannot afford to send the database for all H_i. Instead, in the rLDC setting, Bob enumerates over all possible choices of $DB_1|_{H_i}$ and checks a certain consensus condition. Furthermore, in the rLDC setting, there is no notion of "writes," as encodings of different messages should have large Hamming distance. As a result, there's no equivalent of a DB_0 that makes sense for the rLDC setting, as there are no local ways to update the codeword. On the other hand, rLDCs are secure against all

[5] Note that the assumption that $m \leq \mathsf{poly}(n)$ is somewhat without loss of generality. Indeed, any construction where $m = n^{\omega(1)}$ can be generically transformed into a construction where the physical database size is $\mathsf{poly}(n)$ (by "hashing" the memory space via a pseudorandom function). See Lemma 4 for more details.

[6] A locally decodable code is an error correcting code that allows for recovery of any particular input bit of the message with a small number of queries to the possibly corrupted codeword. A *relaxed* locally decodable code (rLDC) is one that allows the recovery procedure to output $\perp$ if the codeword is indeed corrupted.

computationally unbounded adversaries that change a certain fraction of the codeword, whereas memory checkers need to be secure only for computationally bounded adversaries that can tamper with the whole memory.

2.2 Lower Bound for Low Write Complexity

The above lower bound shows that if the read query complexity q_r is small (e.g., $O(1)$), then the write query complexity must be large (e.g., $n^{\Omega(1)}$). We show that for *deterministic and non-adaptive* memory checkers, the reverse statement is true with analogous quantitative behavior.

In the previous argument, the reason for the $(q_r q_w w)^{O(q_r)}$ term having $O(q_r)$ in the exponent is due to the need to avoid the set $M \cap W$ for all q_r physical queries that Bob does (per logical read $i \in [n]$). This is what prevents the lower bound from being meaningful when $q_r = \omega(\log n / \log\log n)$.

However, let us shift our perspective: what if instead of requiring the reads to avoid the (fixed) writes, we ensure that the writes avoid the reads? Since the memory checker is deterministic and non-adaptive, for all $i \in [n]$, let $W_i, R_i \subseteq [m]$ denote the set of physical locations queried for logical write i and logical read i, respectively. We can define a distribution over $[m]$ as follows, now with respect to the *writes* instead of the reads: sample $i \in [n]$ uniformly randomly, and output a uniform element of W_i. Just as above, we can use the partition lemma to decompose this into H, M, L with respect to this distribution, with $c = 10q_w$ (instead of $c = \Theta(q_r)$). Roughly speaking, this will turn the $(q_r q_w w)^{O(q_r)}$ term from before into a $(q_r q_w w)^{O(q_w)}$ term, which will give us the desired trade-off for low writes.

Now, we can ensure that the writes avoid M: by determinism and non-adaptivity, we know that at least $9/10$ fraction of writes avoid M. Therefore, we can restrict Alice and Bob to consider only these indices when doing the communication argument. Just as before, Alice can send $DB_1|_H$, and all that is left to argue is how to deal with the light writes, i.e., $W \cap L$. By a counting argument, since the queries are light, one can show that the average number of reads that each W_i touches is at most $\delta q_w q_r n$. Therefore, by Markov's inequality, at most a $1/10$ fraction of write indices will touch greater than $10\delta q_w q_r n$ reads (in L). Once again, we can restrict our memory checker to ignore these bad logical indices, leaving $8n/10$ logical indices remaining. Furthermore, among these remaining indices, writing to $k = \Theta(1/(\delta q_w q_r))$ logical indices will leave most reads avoiding $W \cap L$. The communication analysis then proceeds as in the previous proof, arriving at

$$p > \frac{n}{(q_r q_w w)^{O(q_w)}}.$$

In particular, if $p \leq n^{1-\Omega(1)}$ and $q_w = O(1)$ (and $w \leq \mathsf{polylog}(n)$), then $q_r = n^{\Omega(1)}$.

Surpassing determinism and non-adaptivity. Our argument above relies on restricting the logical indices of the memory checker to avoid "bad" indices according to our partition lemma. However, if the memory checker were adaptive, it is not clear how to prevent the "bad" indices from changing adaptively (during or after the writes) in exactly such a way as to break our argument. Also, if the memory checker were randomized, it could be possible that writing to *any* $i \in [n]$ hits M with probability $1/10$. In such a case, it is not clear how to avoid requiring soundness $(9/10)^{\Omega(n^{1-\epsilon})}$ in case $k = \Omega(n^{1-\epsilon})$. This is because any single "bad" write could possibly corrupt all reads, destroying the communication argument. Note that this is *not* true for the previous proof: "bad" reads are not a devastating issue and simply result in slightly less information being communicated.

In Appendix C, we give (weak) evidence to say that proving this same lower bound for *randomized or adaptive* memory checkers might require new ideas. Essentially, we give a generic way to convert memory checkers into new ones that are adaptive and have very cheap writes, at the cost of worse reads.

3 Preliminaries

For a natural number $n \in \mathbb{N}$, we let $[n] := \{k \in \mathbb{N} : 1 \leq k \leq n\}$. We say a function $f : \mathbb{N} \to \mathbb{N}$ is *negligible* if for all constants $c \geq 0$, $\lim_{n\to\infty} f(n) \cdot n^c = 0$. All logarithms are in base 2 unless otherwise specified.

For a distribution $\mathcal{D}$, we write $X \sim \mathcal{D}$ to denote that the random variable X is distributed according to $\mathcal{D}$. For a finite set S, we abuse notation and write $X \sim S$ to denote that the random variable X is distributed uniformly over S. For an often implicitly defined universe set U and $S \subseteq U$, we let $\overline{S} := U \setminus S$ denote the complement of S. For subsets $A_1, \cdots, A_\ell \subseteq U$, we write $U = A_1 \sqcup \cdots \sqcup A_\ell$ to indicate that $\{A_i\}_{i\in[\ell]}$ partition U, i.e., $U = \cup_{i\in[\ell]} A_i$ and $A_i \cap A_j = \emptyset$ for all $i, j \in [\ell], i \neq j$. For a string $x \in \{0,1\}^\ell$, we let $\|x\|_0$ denote the Hamming weight of x, i.e., $\|x\|_0 = |\{i \in [\ell] : x_i = 1\}|$. For strings $x_1 \in \{0,1\}^{\ell_1}$ and $x_2 \in \{0,1\}^{\ell_2}$, we let $x_1||x_2 \in \{0,1\}^{\ell_1+\ell_2}$ denote the concatenation of the two strings. We also use the notation (x_1, x_2) to denote $x_1||x_2$.

Compression lemma. We state the well-known lemma typically used in compression arguments. The lemma says that any method for encoding a uniformly random ℓ-bit string with fewer than ℓ bits must lose information about the string.

Lemma 2 (E.g., [DTT10])**.** *Let S be a finite set. Consider the one-way communication problem of sending uniformly random $x \sim S$ from Alice to Bob in the public coin setting where the length of Alice's message to Bob is fixed. If Bob succeeds in outputting x with probability δ (over x and the public coins), then Alice's (binary) message to Bob must be at least* $\log_2(|S|) - \log_2(1/\delta)$ *bits long.*

3.1 Memory Checking

Here, we define memory checkers for RAMs. Our definitions largely follow classical definitions in the literature (e.g., [BEG+94, NR09, DNRV09]) and below we shall also discuss several distinguishing points.

Random-access machines. A RAM is an interactive Turing machine that consists of a remote memory and a user. The memory is indexed by the logical address space $[n]$. We refer to each memory word also as a block and we use w_ℓ to denote the bit-length of each block. The user can issue read/write instructions to the memory of the form $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$, where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [n]$ and $\mathsf{data} \in \{0,1\}^{w_\ell} \cup \{\bot\}$. If $\mathsf{op} = \mathsf{read}$, then $\mathsf{data} = \bot$ and the returned value is the content of the block located in logical address addr in the memory. If $\mathsf{op} = \mathsf{write}$, then the memory data in logical address addr is updated to data. The user can perform arbitrary computation.

Memory checkers. A memory checker [BEG+94, NR09, DNRV09], at a high level, is a simulation of a standard RAM so that it has the additional guarantee that the RAM's answers to read instructions are reliable even if the remote memory is being tampered with. We formalize a memory checker as an additional interactive Turing machine, placed "between" the user and the remote memory (so they no longer interact directly). The memory checker gets logical queries to a size n memory from the user and processes them into a—*possibly adaptively generated and using randomness*—sequence of read/write instructions to a physical memory. Responses are sent back to the memory checker that processes them until it finally sends a response to the user (if needed). We emphasize that the user issues instructions for the *logical* memory but the simulation is done using a *physical* memory that could possibly be bigger. We let $[m]$ be the address space of the physical memory and assume each word is of size w bits. What makes the task of designing a memory checker non-trivial is the fact that we wish to limit the local space of the memory checker, where by "local space" we mean the number of bits preserved in between user instructions (so it is not a true space complexity parameter, as the memory checker could use an unbounded number of bits in the middle of carrying out a user's query).

Formally, the interface of a memory checker is as follows. We assume that at the start of the system, the local state of the memory checker is empty $\mathsf{st} = \bot$ and the physical memory DB is initialized with all $\bot$s. Furthermore, the memory checker is parametrized by logical and physical memory size and block size (n, w_ℓ) and (m, w), respectively, as well as possibly a computational security parameter given in unary. We suppress these for convenience when clear from context.

- $\langle(\mathsf{data}', \mathsf{st}'), DB'\rangle \leftarrow (\mathsf{MemCheckerOp}(\mathsf{st}, \mathsf{op}, \mathsf{addr}, \mathsf{data}) \iff DB)$. This interactive protocol between the memory checker with local state $\mathsf{st} \in \{0,1\}^p$ and the physical memory $DB \in (\{0,1\}^w \cup \{\bot\})^m$ is executed upon a logical instruction $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$, where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [n]$, and $\mathsf{data} \in \{0,1\}^{w_\ell} \cup \{\bot\}$. The output of this protocol consists of a data entry $\mathsf{data}' \in \{0,1\}^{w_\ell} \cup \{\bot\}$ (where $\mathsf{data}' = \bot$ if $\mathsf{op} = \mathsf{write}$ or if the memory checker detects tampering) and the new local state of the memory checker st', where $|\mathsf{st}'| = p$. The updated physical memory, after the interaction ends, is denoted DB'.

  The protocol may consist of multiple rounds, where in each round the memory checker issues a "physical" RAM query of the form $(\widehat{\mathsf{op}}, \widehat{\mathsf{addr}}, \widehat{\mathsf{data}})$, where $\widehat{\mathsf{op}} \in \{\mathsf{read}, \mathsf{write}\}$, $\widehat{\mathsf{addr}} \in [m]$, and $\widehat{\mathsf{data}} \in \{0,1\}^w$. For every such query, the physical memory DB responds with a value $\widehat{\mathsf{data}}' \in \{0,1\}^w \cup \{\bot\}$, where $\widehat{\mathsf{data}}' = DB[\widehat{\mathsf{addr}}]$ if $\widehat{\mathsf{op}} = \mathsf{read}$ and $\widehat{\mathsf{data}}' = \bot$ otherwise.

The main complexity measure of a memory checker is the communication complexity of the protocols, measured by the total number of queries issued by the memory checker per logical instruction. This is made precise in the following definition.

Definition 1 (Query complexity). *We define the* read query complexity*, denoted q_r (and* write query complexity*, denoted q_w) to be the worst-case number of physical queries made by the memory checker for any given logical read (and logical write, respectively). The* query complexity*, denoted q, is defined as $q = \max\{q_r, q_w\}$.*

Remark 2 (Parameters convention). *We shall assume that all of the above procedures/protocols can be implemented by machines that run in polynomial time in n, the logical memory size. This implicitly bounds $m \leq 2^{\mathsf{poly}(n)}$, as physical indices need $\log m$ bits to represent. If unspecified, we assume that m is bounded by a polynomial in n. We remark that this is (essentially) without loss of generality because memory checkers with a fixed number of supported logical queries, can be generically transformed (via hashing) into ones where $m \in \mathsf{poly}(n)$; see Appendix B for details. Also, unless otherwise specified, we assume $w_\ell, w \leq \mathsf{polylog}(n)$.*

Completeness and soundness. Loosely speaking, completeness of a memory checker says that in an honest execution, i.e., without an adversary tampering with the memory, the user should get the "correct" answer. Soundness says that if the adversary is actively trying to tamper with the memory, then the user will either get the "correct" answer or an abort symbol $\bot$ indicating that something went wrong. Importantly, the memory checker should not respond with a wrong value. Below, we formalize these properties.

We first define the ideal memory functionality, i.e., a memory that is executed by a trusted party and can provide with the "correct" value of every memory cell at any point in time. We refer to this functionality as the *logical memory snapshots functionality.*

Definition 2 (Logical memory snapshot functionality)**.** *A logical memory snapshot functionality is an ideal primitive that gets as input logical instructions and outputs the full state (so called "snapshot") of the logical memory after each instruction. Initially the memory is assumed to be empty, indicated by $\bot$ in each cell. More precisely, the functionality receives (adaptively) a stream of logical instructions of the form* $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$*, where* $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$*,* $\mathsf{addr} \in [n]$*, and* $\mathsf{data} \in \{0,1\}^{w_\ell}$*. Each* read *instruction is ignored. Each* write *instruction is executed, namely, for* $(\mathsf{write}, \mathsf{addr}, \mathsf{data})$ *the value in index* addr *is updated to* data*. After processing each instruction, a snapshot of the memory is generated, denoted* SnapshotMem*.*

For a sequence of l logical instructions $\{(\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i)\}_{i\in[l]}$*, fed into the ideal functionality, we have l snapshots* $\mathsf{SnapshotMem}_1, \ldots, \mathsf{SnapshotMem}_l$*. The ith snapshot satisfies for every* $\mathsf{addr} \in [n]$ *that* $\mathsf{SnapshotMem}_i[\mathsf{addr}] = \mathsf{data} \in \{0,1\}^{w_\ell} \cup \{\bot\}$ *if there exists a $j < i$ such that* $(\mathsf{op}_j, \mathsf{addr}_j, \mathsf{data}_j) = (\mathsf{write}, \mathsf{addr}, \mathsf{data})$ *and there is no $j < j' < i$ with* $(\mathsf{op}_{j'}, \mathsf{addr}_{j'}, \mathsf{data}_{j'}) = (\mathsf{write}, \mathsf{addr}, \mathsf{data}')$ *where* $\mathsf{data}' \neq \mathsf{data}$*.*

Now, we can define completeness. We require that for a fixed instruction index i, with probability $c(n)$ over the internal randomness of the memory checker, the memory checker's output on the ith query is correct (i.e., equal to the corresponding value in the ith logical memory snapshot). Furthermore, we do not allow the logical queries to be adaptively chosen based on previous physical queries made by the memory checker. We emphasize that this definition of completeness is rather weak, and specifically, it is weaker than definitions that were formalized in prior works. However, since we prove a lower bound (meaning we rule out constructions satisfying a weak notion of completeness), these weakenings only make our result stronger. In our upper bound (Section 7), we shall obtain perfect completeness ($c(n) = 1$), achieving the strongest definition of completeness.

Definition 3 (c-completeness)**.** *Let $l \in \mathbb{N}$ and consider a sequence of l logical instructions* $I = \{(\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i)\}_{i\in[l]}$*, where* $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$*,* $\mathsf{addr} \in [n]$*, and* $\mathsf{data} \in \{0,1\}^{w_\ell}$*.*

We say that a memory checker has completeness c if the following holds for every polynomial $l = l(n)$ and I as above:

$$\forall i \in [l]: \ \Pr[\mathsf{CompletenessExpt}(I, i) = 1] \geq c(n),$$

where the probability is over the randomness used in the experiment $\mathsf{CompletenessExpt}(I, i)$ *that is defined next.*

$\underline{\mathsf{CompletenessExpt}(I, i)}$*:*

1. *Initialize a memory checker by setting the local state to* $\mathsf{st} = \bot^p$*, and initialize the physical memory DB to* $\bot^m$*.*

2. *For $j = 1, \ldots, l$:*
   (a) *Run the protocol between the memory checker and the physical memory*
   $$\mathsf{MemCheckerOp}(\mathsf{st}, \mathsf{op}_j, \mathsf{addr}_j, \mathsf{data}_j) \Longleftrightarrow DB.$$
   This protocol outputs a value $\mathsf{data}'_j \in \{0,1\}^{w_\ell} \cup \{\perp\}$. It also outputs an updated st' and an updated physical memory DB', both of which are used in the next iteration.
   (b) *Feed $(\mathsf{op}_j, \mathsf{addr}_j, \mathsf{data}_j)$ into the logical memory snapshot functionality, getting back a snapshot $\mathsf{SnapshotMem}_j$.*
   (c) *If $j = i$, $\mathsf{op}_j = \mathsf{read}$, and $\mathsf{data}'_j \neq \mathsf{SnapshotMem}_j[\mathsf{addr}_j]$, output 0 and abort.*
3. *Output* 1.

Remark 3 (Completeness amplification)**.** *Our definition of completeness allows for amplification by repetition. That is, by running k independent instances of a memory checker at once (sequentially within each logical query) and outputting the majority response, we can improve completeness exponentially in k. This comes at the cost of an $O(k)$ multiplicative blowup in query complexity, local space, and public database size.*

For soundness, we consider a malicious PPT adversary playing the role of the physical memory in the interaction with the memory checker. We require that for any such adversary, and for any given logical read, the probability that the memory checker outputs an *incorrect* value $\sigma \in \{0,1\}^{w_\ell}$ for that read is at most $s(n)$. We emphasize that this is a rather weak notion of soundness and specifically it is weaker than definitions that were formalized in prior works. However, since we prove a lower bound (meaning we rule out constructions satisfying a weak notion of completeness), this only makes our result stronger. In our upper bound (Section 7), we shall prove that our construction satisfies a much stronger soundness guarantee we refer to as *strong* soundness.

Definition 4 (s-soundness)**.** *Let $l \in \mathbb{N}$ and consider a sequence of l logical instructions $I = \{(\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i)\}_{i \in [l]}$, where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [n]$, and $\mathsf{data} \in \{0,1\}^{w_\ell}$.*

We say that a memory checker has soundness $s(n)$ if for every stateful PPT adversary $\mathcal{A}$ and every polynomial $l = l(n)$ and I as above, it holds that

$$\forall i \in [l]\colon \; \Pr[\mathsf{SoundnessExpt}_{\mathcal{A}}(I, i) = 1] \leq s(n),$$

where the probability is over the randomness used in the experiment $\mathsf{SoundnessExpt}_{\mathcal{A}}(I, i)$ that is defined next.

<u>$\mathsf{SoundnessExpt}_{\mathcal{A}}(I, i)$</u>*:*

1. *Initialize a memory checker by setting the local state to $\mathsf{st} = \perp^p$.*
2. *For $j = 1, \ldots, l$:*
   (a) *Run the protocol between the memory checker and $\mathcal{A}$, acting as the physical memory:*
   $$\mathsf{MemCheckerOp}(\mathsf{st}, \mathsf{op}_j, \mathsf{addr}_j, \mathsf{data}_j) \Longleftrightarrow \mathcal{A}.$$
   The adversary $\mathcal{A}$ has read access to everything except for the local state of the memory checker and the random tape of the memory checker. This protocol outputs a value $\mathsf{data}'_j \in \{0,1\}^{w_\ell} \cup \{\perp\}$. It also outputs an updated st' and an updated physical memory DB', both of which are used in the next iteration.

(b) *Feed* $(\mathsf{op}_j, \mathsf{addr}_j, \mathsf{data}_j)$ *into the logical memory snapshot functionality, getting back a snapshot* $\mathsf{SnapshotMem}_j$*.*

(c) *If* $j = i$*,* $\mathsf{op}_j = \mathsf{read}$*, and* $\mathsf{data}'_j \notin \{\mathsf{SnapshotMem}_j[\mathsf{addr}_j], \perp\}$*, output* 1 *and abort.*

3. *Output* 0.

Remark 4 (Completeness and soundness parameters)**.** *In constructions of memory checkers it is desired to have* c *as close to* 1 *as possible and* s *as small as possible. Usually, we have perfect completeness (i.e.,* $c = 1$*) and negligible soundness (in a security parameter). These are also the guarantees of our construction in Section 7. In our lower bounds, however, we rule out a much weaker memory checker (which makes our lower bound stronger), where completeness is* $c = 2/3$ *and soundness is inverse polynomial in* n*.*

We now define a stronger version of soundness, which we call *strong* soundness. There are three differences between this notion and soundness as defined above. First, strong soundness allows the adversary to view (but not modify) the local state and random tape of the memory checker. Second, strong soundness allows the logical queries to be adaptively chosen by the adversary instead of fixed in advance. Third, an adversary wins the strong soundness game if it causes the memory checker to err on *any* logical query, as opposed to a specific logical instruction fixed in advance.

We now give a formal definition.

Definition 5 (s-strong-soundness)**.** *We say that a memory checker has strong soundness* $s(n)$ *if for every stateful PPT adversary* $\mathcal{A}$*, it holds that*

$$\Pr[\mathsf{StrongSoundnessExpt}_{\mathcal{A}}() = 1] \leq s(n),$$

where the probability is over the randomness used in the experiment $\mathsf{StrongSoundnessExpt}_{\mathcal{A}}()$ *that is defined next.*

$\underline{\mathsf{StrongSoundnessExpt}_{\mathcal{A}}()}$*:*

1. *Initialize a memory checker by setting the local state to* $\mathsf{st} = \perp^p$*.*

2. *While* $(\mathsf{op}, \mathsf{addr}, \mathsf{data}) \leftarrow \mathcal{A}(\mathsf{st})$*:*

   (a) *Run the protocol between the memory checker and* $\mathcal{A}$*, acting as the physical memory:*

   $$\mathsf{MemCheckerOp}(\mathsf{st}, \mathsf{op}, \mathsf{addr}, \mathsf{data}; r) \Longleftrightarrow \mathcal{A}(\mathsf{st}, r),$$

   where here, r *denotes the random tape of the memory checker. We emphasize that that* $\mathcal{A}$ *has read access to everything, including the local state and the random tape of the memory checker. This protocol outputs a value* $\mathsf{data}' \in \{0,1\}^{w_\ell} \cup \{\perp\}$*. It also outputs an updated* st' *and an updated physical memory* DB'*, both of which are used in the next iteration.*

   (b) *Feed* $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ *into the logical memory snapshot functionality, getting back an updated snapshot* $\mathsf{SnapshotMem}$*.*

   (c) *If* $\mathsf{op} = \mathsf{read}$ *and* $\mathsf{data}' \notin \{\mathsf{SnapshotMem}[\mathsf{addr}], \perp\}$*, output* 1 *and abort.*

3. *Output* 0.

We now define what it means for a memory checker to be *deterministic and non-adaptive*, in the sense of Dwork et al. [DNRV09]. Recall that Dwork et al. [DNRV09] give a lower bound only for deterministic and non-adaptive memory checkers, while our main lower bound makes no such restriction.

Definition 6 (Deterministic and non-adaptive memory checker)**.** *A memory checker is* deterministic and non-adaptive *if there exist fixed sequences $W_1, \cdots, W_n \in [m]^{q_w}$ and $R_1, \cdots, R_n \in [m]^{q_r}$ such that each logical write (or read) to index $i \in [n]$ performs physical queries at the physical locations exactly according to the sequence W_i (or R_i, respectively), regardless of the local state, contents on the database, or random coins. Note that the contents of the queries can still be randomized and adaptive; it is just the physical locations queried that are completely fixed.*

Comparison with prior definitions. In comparing to other definitions of memory checking, our lower bound applies to the widest class of constructions while not restricting the adversary (apart from being computationally efficient). Besides the differences mentioned above, we mention a few other ways in which our lower abound applies more broadly.

Since we define the completeness (and soundness in Definition 4) game to succeed for some particular instruction index $i \in [l]$, it is possible that a memory checker with completeness $2/3$ makes *some* mistake with high probability over $\ell(n) = n$ queries. For example, suppose that the memory checker is independently wrong for each read with probability $1/3$. Then, this would still satisfy $2/3$ completeness, but the probability that all outputs are correct is $1/2^{\Omega(n)}$. Prior definitions of memory checking would consider this memory checker to have "completeness" $1/2^{\Omega(n)}$. Another way in which our lower bound is more broad is that it applies to memory checkers that could have *secret* local state. Furthermore, our lower bound applies to memory checkers that support $2n$ logical queries (and in fact, just one logical read), as opposed to an arbitrary polynomial number of logical queries. Moreover, our lower bound applies even when $w_\ell = 1$. Lastly, our main lower bound does not require the memory checker's internal operations to be efficiently computable. In fact, the local space is just the number of bits preserved *in between* user queries, so it is not a true space complexity parameter, as the memory checker could use an unbounded number of bits in the middle of carrying out a user query.

3.2 Cryptographic Primitives

In this section, we define pseudorandom functions (PRFs) [GGM86] and universal one-way hash functions (UOWHFs) [NY89].

Definition 7 (PRF)**.** *Let λ denote the security parameter. We say that a family of functions $\mathcal{F} = \{\mathsf{PRF}_k : \{0,1\}^\lambda \to \{0,1\}^\lambda\}_{k \in \{0,1\}^\lambda}$ is a* pseudorandom function *(PRF) family if the following two properties hold:*

- ***Efficiency**: Given $k \in \{0,1\}^\lambda$ and $x \in \{0,1\}^\lambda$, $\mathsf{PRF}_k(x) \in \{0,1\}^\lambda$ can be computed in $\mathsf{poly}(\lambda)$ time.*
- ***Pseudorandomness**: For all PPT adversaries $\mathcal{A}$,*

$$\left| \Pr_{k \sim \{0,1\}^\lambda}[\mathcal{A}^{\mathsf{PRF}_k(\cdot)}(1^\lambda) = 1] - \Pr_{F \sim \mathsf{All}}[\mathcal{A}^{F(\cdot)}(1^\lambda) = 1] \right| \leq \mathsf{negl}(\lambda),$$

where $F \sim \mathsf{All}$ denotes sampling F uniformly at random from the set of all functions $\{0,1\}^\lambda \to \{0,1\}^\lambda$.

We note that PRFs exist if one-way functions exist [HILL99, GGM86]. Furthermore, we note that we can easily extend the domain and codomain of the PRF to $\{0,1\}^{\mathsf{poly}(\lambda)}$ by standard transformations (e.g., see [Gol01]).

Definition 8 (UOWHF)**.** *Let λ denote the security parameter. We say that a family of functions $\mathcal{H} = \{H_k : \{0,1\}^{\lambda^2} \to \{0,1\}^\lambda\}_{k \in \{0,1\}^\lambda}$ is a family of $(T(\lambda), \epsilon(\lambda))$-secure* universal one-way hash functions *(UOWHFs) if the following hold:*

- ***Efficiency**: Given $k \in \{0,1\}^\lambda$ and $x \in \{0,1\}^{\lambda^2}$, $H_k(x) \in \{0,1\}^\lambda$ can be computed in $\mathsf{poly}(\lambda)$ time.*
- ***Target Collision Resistance**: For all (stateful) adversaries $\mathcal{A}$ running in time $T(\lambda)$, the following holds:*

$$\Pr\left[\mathcal{A}(1^\lambda) \to (x, \mathsf{st}); k \sim \{0,1\}^\lambda; \mathcal{A}(1^\lambda, k, \mathsf{st}) \to x' : H_k(x) = H_k(x') \wedge x \neq x'\right] \leq \epsilon(\lambda).$$

If $\mathcal{H}$ is $(p_1(\lambda), 1/p_2(\lambda))$-secure for all polynomials $p_1(\lambda)$ and $p_2(\lambda)$, then we say $\mathcal{H}$ is polynomially secure*. If there exists some $\delta \in (0,1)$ such that $\mathcal{H}$ is $(2^{\lambda^\delta}, 1/2^{\lambda^\delta})$-secure, then we say $\mathcal{H}$ is* sub-exponentially secure*.*

We note that polynomially secure UOWHFs exist if one-way functions exist, and similarly, sub-exponentially secure UOWHFs exist if sub-exponentially secure one-way functions exist [Rom90, KK05]. Furthermore, we note that extending the domain of the hash function to $\{0,1\}^{\lambda^2}$ (instead of, say, $\{0,1\}^{2\lambda}$ or $\{0,1\}^{\lambda+1}$) can be done using a standard UOWHF domain extension techniques [BR97].

4 Main Lower Bound

We begin by describing our main technical theorem:

Theorem 5 (Main Theorem)**.** *Consider a memory checker for a logical memory of size n and with logical word size $w_\ell = 1$, supporting $2n$ logical queries. Assume that the physical database is of size m and with physical word size w, the read query complexity is q_r, the write query complexity is q_w, and the local space is p. If completeness is $99/100$ and soundness is $1/(10^4 \cdot n \cdot q_r \cdot 2^{q_r})$, then there is a universal constant $C \leq 10^3$ such that for $n \geq C$,*

$$p \geq \frac{n}{(Cq_rq_w(w + \log m))^{C \cdot q_r}} - \log q_r - C.$$

Next, we state a couple of corollaries of this theorem. These corollaries, together with Theorem 5, are the precise and elaborate versions of Theorems 1 and 3 from the introduction. The first corollary gives a (quasi-)logarithmic lower bound on the worst-case query complexity of every memory checker where the reads and writes cost the same. The second corollary allows us to conclude that if the read query complexity is small, then the write complexity must be large. We show how these corollaries follow from Theorem 5 below, followed by a proof of Theorem 5.

Corollary 1. *In the setting of Theorem 5 and further assuming that $q = \max\{q_w, q_r\}$, $m \leq \mathsf{poly}(n)$, and $w \leq \mathsf{polylog}(n)$, if completeness is $2/3$ and soundness is $1/n^{1+o(1)}$, it holds that*

$$p \geq \frac{n}{(\log n)^{O(q)}} - O(\log \log n).$$

In particular, if $p < n^{1-\varepsilon}$ for some $\epsilon > 0$, then $q \geq \Omega(\log n / \log \log n)$.

Corollary 2. *In the setting of Theorem 5 and further assuming that $m \leq \mathsf{poly}(n)$, $w \leq \mathsf{polylog}(n)$, and $p \leq n^{1-\varepsilon}$ for some $\varepsilon > 0$, if completeness is $2/3$ and soundness is $1/n^{1+o(1)}$, it holds that,*

- *If $q_r = o(\log n / \log \log n)$, then $q_w = (\log n)^{\omega(1)}$.*
- *If $q_r = O(1)$, then $q_w = n^{\Omega(1)}$.*

We first prove that Corollary 1 follows from Theorem 5.

Proof of Corollary 1 assuming Theorem 5. First, we apply completeness amplification by running $\Theta(1)$ independent instances at once, to get a memory checker with completeness $99/100$, read- and write-query complexities $q_r' = \Theta(q_r)$ and $q_w' = \Theta(q_w)$ (so $q' = \max\{q_r', q_w'\} = \Theta(q)$), local space $p' = \Theta(p)$, and $m' = \Theta(m) \leq \mathsf{poly}(n)$. Suppose that $q' \geq \log(n)/\log\log(n)$. Then, there exists a constant C such that $(\log n)^{C \cdot q} \geq n$, in which case the result is trivial as the right hand side is negative. Therefore, we can assume throughout that we are in the case where $q_r', q_w' \leq \log(n)/\log\log(n)$. This implies that $\log q_r' \leq O(\log \log n)$ and $1/(10^4 \cdot n \cdot q_r' \cdot 2^{q_r'}) = 1/n^{1+o(1)}$. We then apply Theorem 5 with $m' \leq \mathsf{poly}(n)$, and thus $\log(m') \leq O(\log n)$. This makes the denominator $(\log n)^{O(q)}$ and the result follows. □

Second, we prove that Corollary 2 follows from Theorem 5.

Proof of Corollary 2 assuming Theorem 5. We similarly apply completeness amplification by running $\Theta(1)$ independent instances at once, to get a memory checker with completeness $99/100$, read- and write-query complexities $q_r' = \Theta(q_r)$ and $q_w' = \Theta(q_w)$, local space $p' = \Theta(p)$, and $m' = \Theta(m) \leq \mathsf{poly}(n)$. This implies $\log(m') = O(\log n)$. In both cases we need to prove, we have $q_r' = \Theta(q_r) = o(\log n / \log \log n)$. Applying Theorem 5, we get

$$p' + \log q_r' + C \geq \frac{n}{(q_r q_w \log n)^{C q_r}}$$

for some universal constant C. Since $p \geq n^{1-\varepsilon}$ for some $\varepsilon > 0$ and since $\log q_r'$ is a lower order term, we have

$$\frac{n}{(q_r q_w \log n)^{\Theta(q_r)}} \leq n^{1-\varepsilon}.$$

Rearranging, this gives

$$q_w \geq \frac{n^{\varepsilon/\Theta(q_r)}}{q_r \log n}.$$

If $q_r = O(1)$, then $q_w \geq n^{\Omega(1)}$, and if $q_r = o(\log n / \log \log n)$, then $q_w \geq (\log n)^{\omega(1)}$, as needed. □

The proof of Theorem 5 appears in Section 4.1.

Remark 5 (On super-polynomial public database size)**.** *It makes sense to assume that m is bounded by some a priori unspecified polynomial in n (i.e., $m \leq \mathsf{poly}(n)$, as we assume in Corollaries 1 and 2), and in this case the lower bounds from Theorem 5 and Corollaries 1 and 2 are unconditional. But what if m is super-polynomial in n? Interestingly, we can still get a meaningful result. First, one can generically reduce the public database size from $m \leq 2^{\mathsf{poly}(n)}$ to $m \leq \mathsf{poly}(n)$ in any memory checker construction, by hashing the address space using a PRF (see Lemma 4 for details). This transformation, on its own, relies on the existence of one-way functions, but we recall that a memory checker satisfying a non-trivial relationship between p and q (i.e., $p \cdot q = o(n)$) already implies the existence of (infinitely often) one-way functions [NR09] which in turn can be used to get an (infinitely often) PRF [HILL99, GGM86].*

4.1 Proof of Theorem 5

As explained in the technical overview, the proof proceeds by a compression argument where we utilize a memory checker to convey information from Alice to Bob. Furthermore, the main technical tool that we use is a partition lemma that allows us to classify and split physical locations into heavy, medium, and light. We first state and prove the partition lemma inspired by [Gol23, Claim 2.6] and then proceed with the main proof.

Lemma 3. *Let X be a random variable supported on a finite set S. Let $\gamma > 1$, and let $c, n \in \mathbb{N}$. Then, there is a partition $S = L \sqcup M \sqcup H$ such that the following hold for some $\delta \geq 1/n$:*

- $\Pr[X = \ell] \leq \delta$ *for all $\ell \in L$,*
- $\Pr[X \in M] \leq 1/c$, *and*
- *The set H satisfies*
$$\frac{1}{\delta} - |H|\gamma > \frac{n}{(2\gamma)^c}.$$

Moreover, there exists $i \in [c]$ such that the conditions above hold for $\delta = (2\gamma)^{i-1}/n$.

Looking ahead, we use this last property of δ to argue that for fixed γ, c and n, one can guess a valid value of δ with probability $1/c$ without knowing the distribution of X.

Proof of Lemma 3. We define the sets

$$\begin{aligned}
B_1 &:= \left\{ s \in S : \Pr[X = s] \in \left[0, \frac{2\gamma}{n}\right) \right\}, \\
B_i &:= \left\{ s \in S : \Pr[X = s] \in \left[\frac{(2\gamma)^{i-1}}{n}, \frac{(2\gamma)^{i}}{n}\right) \right\} \text{ for } i \in \{2, \cdots, c-1\}, \\
B_c &:= \left\{ s \in S : \Pr[X = s] \in \left[\frac{(2\gamma)^{c-1}}{n}, \infty\right) \right\},
\end{aligned}$$

where we have $S = \bigsqcup_{i \in [c]} B_i$ by construction. By an averaging argument, we know there must exist some particular index $j \in [c]$ such that $\Pr[X \in B_j] \leq 1/c$. We then set $L = \bigcup_{i \leq j-1} B_i$, $M = B_j$, and $H = \bigcup_{i \geq j+1} B_i$. Clearly, $\Pr[X \in M] \leq 1/c$ by construction.

Since for all $h \in H$ we have $\Pr[X = h] \geq (2\gamma)^j/n$, by summing over $h \in H$, we know $|H| \leq n/(2\gamma)^j$. On the other hand, by definition of L, we know that for all $\ell \in L$, $\Pr[X = \ell] < (2\gamma)^{j-1}/n$. (Note that this is vacuously true for $j = 1$.) We have $(2\gamma)^{j-1}/n \geq 1/n$ since $\gamma > 1$ and $j \geq 1$, so we can set $\delta = (2\gamma)^{j-1}/n$. Combining the inequalities above, we have

$$\frac{1}{\delta} - |H|\gamma \geq \frac{n}{(2\gamma)^{j-1}} - \frac{n}{(2\gamma)^j} \cdot \gamma = \frac{n}{2 \cdot (2\gamma)^{j-1}} \geq \frac{n}{2 \cdot (2\gamma)^{c-1}} > \frac{n}{(2\gamma)^c},$$

as desired. □

We now proceed with the proof of the main theorem, Theorem 5. We start by setting a few parameters in anticipation of applying the partition lemma (Lemma 3) to a distribution over physical memory locations. Let $c = 100q_r$ and $\gamma = 200q_rq_w(w + \log m + 2)$ as needed for Lemma 3. For $j \in [c]$, let $\delta_j := (2\gamma)^{j-1}/n$ as given by Lemma 3, and let $k_j := \lfloor 1/(100\delta_j q_r q_w) \rfloor$, where one should think of k_j (for $j \in [c]$) as the number of 1s written to the memory checker. Later on in the proof, the choice of these parameter settings will become more clear. (Specifically, c and k_j are set by Claim 2, and γ is set at the end of the proof.) Observe that since $\delta_j \geq 1/n$, we have $k_j \leq 1/(100\delta_j) \leq n/100$. Throughout, we assume n is a multiple of 10 for simplicity, but our proof can be easily modified to handle arbitrary, sufficiently large n.

Using the memory checker, we directly construct a public-coin protocol for Alice and Bob to send c strings $x^{(1)}, \cdots, x^{(c)}$, distributed as follows: for each $j \in [c]$, $x^{(j)} \in \{0,1\}^{9n/10+k_j}$ is independently chosen uniformly at random subject to the constraint that $\|x^{(j)}\|_0 = k_j$ (i.e., the Hamming weight of each $x^{(j)}$ is k_j). At a high level, the size of Alice's message will be closely related to the local space p of the memory checker, so success of this protocol will yield a lower bound on p.

Protocol description. Before Alice sees $x^{(1)}, \cdots, x^{(c)}$, Alice and Bob together (using shared randomness) run a memory checker and logically write 0 to all indices $i \in [n]$. Then, Alice and Bob together sample $j^* \sim [c]$ independently and uniformly at random. We urge the reader to think of j^* as a guess for $j \in [c]$ for which $\delta = (2\gamma)^{j-1}/n$ will appear when applying Lemma 3 later on in the protocol. Since the memory checker could be adaptive, we may not know the right value of j beforehand, so we guess it. For simplicity, on a first pass, one may think of j as being "fixed" to the right value, although Alice has no way of knowing this beforehand.

Alice and Bob then sample uniformly random $y \in \{0,1\}^n$ with $\|y\|_0 = n/10 - k_{j^*}$ and logically write 1 to all indices $i \in [n]$ such that $y_i = 1$ in a uniformly random order. This gives Alice and Bob the public database $DB_0 \in (\{0,1\}^w \cup \{\perp\})^m$ after these writes as well as the local state $\mathsf{st}_0 \in \{0,1\}^p$. (This can all be formalized by Alice and Bob sharing a sufficiently long uniformly random string and running the memory checker on that string.)

For the remainder of the protocol, Alice and Bob agree on some mapping $\pi : [9n/10 + k_{j^*}] \to [n]$ that maps $[9n/10 + k_{j^*}]$ bijectively to the indices $i \in [n]$ where $y_i = 0$. For notational simplicity, we set $x = x^{(j^*)}$ and $k = k_{j^*}$ for the rest of the proof.

Alice's encoding. In short, for all $j \neq j^*$, Alice will directly encode $x^{(j)}$, but for $j = j^*$, Alice will write 1 to logical indices $\pi(x) := \{\pi(i) : i \in [9n/10 + k], x_i = 1\}$ of the memory checker in a random order and send some information related to the memory checker at the end of the writes.

More precisely, Alice tosses coins and uses the memory checker (with DB_0 and st_0) to write 1 to every logical index in the set $\pi(x)$ in a uniformly random order. This produces a new public database DB_1 and local state st_1. Now, Alice defines a distribution $\mathcal{D}$ over $[m]$ as follows:

1. Sample $i \sim [n]$ uniformly at random.
2. Use the memory checker to read index i from local space st_1 and public database DB_1. This defines a distribution over sequences of length q_r of the physical database locations, corresponding to the locations accessed for logical read i. Sample such a sequence R from this distribution (i.e., $|R| = q_r$).
3. Finally, sample $v \sim R$ uniformly at random and output v.

Alice now applies Lemma 3 to this distribution $\mathcal{D}$ with parameters c and γ to partition the physical locations into $[m] = L \sqcup M \sqcup H$ and getting a parameter δ of the form $\delta = (2\gamma)^{\widetilde{j}-1}/n$ for some $\widetilde{j} \in [c]$ (e.g., choosing the smallest possible $\widetilde{j}$ for which the above holds). If $\widetilde{j} \neq j^*$, Alice aborts and the whole protocol fails (This can be formalized by Alice sending the all 0s string.) See Figure 3 for explicit details.

We make the following claim:

Claim 1. *The random variables j^* and $\widetilde{j}$ are independent. In particular,* $\Pr\left[\widetilde{j} = j^*\right] = 1/c$, *where the probability is over all randomness sampled by Alice and Bob in the protocol.*

Proof of Claim 1. Recall that the marginal distribution of j^* is uniformly random over $[c]$. We now argue that the random variable $\widetilde{j} \in [c]$ is independent of j^*.

The distribution of $\widetilde{j}$ can be described by the following random process, incorporating randomness used internally by the memory checker and by the logical queries passed into the memory checker:

1. Logically write 0 to all indices $i \in [n]$.
2. Sample $j^* \sim [c]$.
3. Logically write 1 to $n/10 - k_{j^*}$ uniformly random distinct indices (denoted by y above) in a uniformly random order.
4. Logically write 1 to k_{j^*} uniformly random distinct indices in a uniformly random order, not overlapping with indices from previous step (denoted by $\pi(x^{(j^*)})$ above).
5. Use the resulting DB_1, st_1 from the previous step to define a distribution $\mathcal{D}$. Apply Lemma 3 to $\mathcal{D}$ to generate $\widetilde{j}$. (This step is deterministic given the previous steps.)

This can be equivalently defined by the following random process:

1. Logically write 0 to all indices $i \in [n]$.
2. Logically write 1 to $n/10$ uniformly random distinct indices in a uniformly random order.
3. Use the resulting DB_1, st_1 from the previous step to define a distribution $\mathcal{D}$. Apply Lemma 3 to $\mathcal{D}$ to generate $\widetilde{j}$. (This step is deterministic given the previous steps.)

The latter random process has no dependence on j^*. Therefore, j^* and $\widetilde{j}$ are independent. $\square$

For the rest of the protocol description, we condition on the event that $\widetilde{j} = j^*$. Thus, Alice sends the following information to Bob:

- A direct encoding of $\left(x^{(1)}, \cdots, x^{(j^*-1)}, x^{(j^*+1)}, \cdots, x^{(c)}\right)$,
- The updated local state st_1,
- A description of $H \subseteq [m]$,
- $DB_1|_H$, i.e., the contents of DB_1 at the locations in H, and
- Some auxiliary string aux (specified later in the proof) which will help Bob complete Alice's information into a full description of x.

Because the physical database has size m, the description of H can be represented using $\binom{m}{|H|} + 1 \leq |H|\lceil\log(m)\rceil$ bits. As such, the total size of this message (in bits) can be upper bounded[7] by

$$\left\lceil \log\left(\prod_{j\in[c], j\neq j^*} \binom{9n/10 + k_j}{k_j}\right)\right\rceil + |\mathsf{st}_1| + |H|\lceil\log(m)\rceil + |H|\lceil\log(2^w+1)\rceil + |\mathsf{aux}|$$

$$\leq \log\left(\prod_{j\in[c], j\neq j^*} \binom{9n/10 + k_j}{k_j}\right) + p + |\mathsf{aux}| + |H|(w + \log m + 2) + 1.$$

Bob's decoding. Given H and $DB_1|_H$ from Alice, Bob can recreate some partially updated public database $\widetilde{DB} \in (\{0,1\}^w \cup \{\perp\})^m$, which is defined as DB_1 on locations in H and DB_0 on $\overline{H} = L \sqcup M$. In short, Bob's strategy will simulate the memory checker on the next logical read to all possible $i \in [n]$, rewinding back each time, using $\widetilde{DB}$ and st_1 from Alice.

More formally, let $r \in \{0,1\}^t$ denote the random bits used by the memory checker (for this final read). Let $MC(i, DB, \mathsf{st}; r) \in \{0, 1, \perp\}$ denote the output of the memory checker upon performing a logical read to index $i \in [n]$ from local state st and public database DB using randomness r. Let $\mathsf{BitMajority}\colon \{0,1,\perp\}^{2^t} \to \{0,1,\perp\}$ be the function that takes in the *bit-wise* majority of the inputs, meaning outputting the majority bit if such a bit exists, and otherwise, outputting $\perp$. For each $i \in [n]$, Bob will set

$$\tilde{z}_i := \mathsf{BitMajority}\left(MC\left(i, \widetilde{DB}, \mathsf{st}_1; r\right)_{r\in\{0,1\}^t}\right).$$

That is, Bob will brute force over all $i \in [n]$ and random bits used by the memory checker for this read, and Bob will deduce a bit value $\tilde{z}_i$ for z_i, the ith logical bit of the memory checker, if there is a strict majority of choices of randomness that agree on a bit (not $\perp$) to output. Otherwise,

[7] As a technicality, as stated, the length of this message is not fixed, but rather is a random variable that depends on the protocol's execution. Our proof, essentially, will show that with sufficiently high probability, this random variable can be upper bounded by a small enough fixed quantity to invoke Lemma 2 and obtain a lower bound on $p = |\mathsf{st}_1|$.

Alice's Strategy

Shared by Alice and Bob: $j^*, DB_0, \mathsf{st}_0, y, \pi$
Alice's Private Input: $\left(x^{(1)}, \cdots, x^{(c)}\right)$

1. Let $x := x^{(j^*)}$, and let $x^{(-j^*)}$ be a direct encoding of $\left(x^{(1)}, \cdots, x^{(j^*-1)}, x^{(j^*+1)}, \cdots, x^{(c)}\right)$.
2. Let $\pi(x) := \{\pi(\ell) \in [n] : \ell \in [9n/10 + k_{j^*}], x_\ell = 1\}$.
3. Using DB_0 and st_0, use the memory checker to write "1" to all indices in $\pi(x) \subseteq [n]$ in a uniformly random order. This results in updated DB_1 and st_1.
4. Using DB_1 and st_1, apply Lemma 3 to the resulting distribution $\mathcal{D}$ to get $[m] = H \sqcup M \sqcup L$ and $\widetilde{j} \in [c]$.
5. If $j^* \neq \widetilde{j}$, immediately abort the whole protocol.
6. Run Steps 2-4 of Bob's strategy (Figure 4) to generate the subset $U \subseteq [n]$.
7. Let aux be an encoding of the subset $U \cap \pi(x) \subseteq U$.

Output: $\left(x^{(-j^*)}, \mathsf{st}_1, H, DB_1|_H, \mathsf{aux}\right)$.

Figure 3: Alice's Encoding Strategy.

Bob will put some placeholder value for the ith bit and use aux from Alice to fill it in. Reading off the values at the indices in $\pi([9n/10 + k]) \subset [n]$ will then yield Bob's output $\tilde{x} \in \{0,1\}^{9n/10+k}$. See Figure 4 for explicit details. Note that Bob's decoding strategy may not be computationally efficient; however, this is not an issue for the information theoretic compression argument.

We now analyze how successful Bob will be. At a high level:

- We use completeness of the memory checker to argue that whenever $\widetilde{DB}$ and DB_1 are consistent for a read (which will happen pretty often per Claim 2 below), then Bob learns z_i.
- We use soundness to argue that Bob never learns the wrong value of *any* z_i (but could get $\perp$ from the memory checker).

This communicates some information from Alice to Bob, which we show is enough to get the desired lower bound. The string aux is used so that Alice can complete Bob's partial information into full information about x.

Recall that DB_0 and DB_1 can differ only at locations that Alice accessed in writing indices $\pi(i)$ where $x_i = 1$. Since $\|x\|_0 \leq k$, this is at most $k \cdot q_w$ locations, which we will call $W \subseteq [m]$. Define $\mathrm{BAD} := W \cap (L \cup M) = W \cap \overline{H}$ to be the set of locations that Alice wrote to that are not included in H. That is, DB_1 and $\widetilde{DB}$ differ only at physical locations in BAD. (We emphasize that Bob does not know the set BAD.)

We now argue that Bob's physical queries avoid BAD with constant probability. Let $R(i, DB, \mathsf{st}; r) \subseteq [m]$ denote the set of (at most q_r) physical locations queried by the memory checker upon performing logical read to index $i \in [n]$ using public database DB, local state st, and internal randomness r.

Bob's Strategy

Shared by Alice and Bob: $j^*, DB_0, \mathsf{st}_0, y, \pi$
Alice's Message: $\left(x^{(-j^*)}, \mathsf{st}_1, H, DB_1|_H, \mathsf{aux}\right)$.

1. Decode $x^{(-j^*)}$ into $\left(x^{(1)}, \cdots, x^{(j^*-1)}, x^{(j^*+1)}, \cdots, x^{(c)}\right)$.
2. Using DB_0 from the shared input and H and $DB_1|_H$ from Alice, define the database $\widetilde{DB} \in (\{0,1\}^w \cup \{\perp\})^m$ as follows:
$$\widetilde{DB}[v] := \begin{cases} DB_1[v] & \text{if } v \in H, \\ DB_0[v] & \text{otherwise.} \end{cases}$$
3. For all $i \in [n]$, let
$$\tilde{z}_i := \mathsf{BitMajority}\left(MC\left(i, \widetilde{DB}, \mathsf{st}_1; r\right)_{r \in \{0,1\}^t}\right) \in \{0, 1, \perp\}.$$
4. Define $U := \{i \in [n] : \tilde{z}_i = \perp\}$.
5. Parsing $\mathsf{aux} \subseteq U$, define $z \in \{0,1\}^n$ by
$$z_i := \begin{cases} \tilde{z}_i & \text{if } \tilde{z}_i \neq \perp, \\ 1 & \text{if } \tilde{z}_i = \perp \text{ and } i \in \mathsf{aux}, \\ 0 & \text{if } \tilde{z}_i = \perp \text{ and } i \notin \mathsf{aux}. \end{cases}$$
6. Define $x \in \{0,1\}^{9n/10 + k_{j^*}}$ by $x_\ell := z_{\pi(\ell)} \in \{0,1\}$.

Output: $\left(x^{(1)}, \cdots, x^{(j^*-1)}, x, x^{(j^*+1)}, \cdots, x^{(c)}\right)$.

Figure 4: Bob's Decoding Strategy.

Claim 2. *Assuming* $\widetilde{j} = j^*$,

$$\Pr_{i \sim [n], r \sim \{0,1\}^t}\left[R\left(i, DB_1, \mathsf{st}_1; r\right) \cap \mathrm{BAD} = \emptyset\right] \geq \frac{98}{100}.$$

Proof. Since $\mathrm{BAD} = (W \cap M) \cup (W \cap L)$, we can argue the two cases separately.

For $X \sim \mathcal{D}$, we know that $\Pr[X \in M] \leq 1/c = 1/(100 q_r)$. By construction of $\mathcal{D}$ and the union bound, this implies

$$\begin{aligned} \Pr_{i \sim [n], r \sim \{0,1\}^t}[R(i, DB_1, \mathsf{st}_1; r) \cap (W \cap M) \neq \emptyset] &\leq \Pr_{i \sim [n], r \sim \{0,1\}^t}[R(i, DB_1, \mathsf{st}_1; r) \cap M \neq \emptyset] \\ &\leq q_r \cdot \Pr_{X \sim \mathcal{D}}[X \in M] \leq \frac{1}{100}. \end{aligned}$$

For the other case, by definition of L and δ_{j^*}, and for $X \sim \mathcal{D}$, we know that

$$\Pr_{X\sim\mathcal{D}}[X \in W \cap L] = \sum_{\ell \in W \cap L} \Pr_{X\sim\mathcal{D}}[X = \ell] \leq |W \cap L|\delta_{j^*}.$$

Thus, by construction of $\mathcal{D}$,

$$\begin{aligned}\Pr_{i\sim[n],r\sim\{0,1\}^t}[R(i, DB_1, \mathsf{st}_1; r) \cap (W \cap L) \neq \emptyset] &\leq q_r \cdot \Pr[X \in W \cap L] \leq q_r\delta_{j^*}|W \cap L| \\ &\leq q_r\delta_{j^*}|W| \leq kq_wq_r\delta_{j^*} \leq \frac{1}{100},\end{aligned}$$

where the last inequality holds since $k = k_{j^*} = \lfloor 1/(100\delta_{j^*}q_rq_w)\rfloor$. By a union bound over both cases, since $\mathrm{BAD} = (W \cap M) \cup (W \cup L)$, we have

$$\Pr_{i\sim[n],r\sim\{0,1\}^t}[R(i, DB_1, \mathsf{st}_1; r) \cap \mathrm{BAD} \neq \emptyset] \leq \frac{1}{100} + \frac{1}{100} = \frac{2}{100}.$$

□

Given the above claim, we can invoke completeness of the memory checker, since avoiding BAD means that the physical locations accessed are correct. Explicitly, we invoke completeness on the sequence of logical queries that writes 0 everywhere, writes 1 to $n/10$ random locations denoted by z, and lastly reads to a random location, denoted by i. We will use completeness for the last logical read, i.e., query number $n + n/10 + 1$.

Let r_1 and r_2 denote the randomness of the memory checker used for all of the logical writes and randomness of the memory checker used for the final logical read, respectively. Let z_i denote the ith logical bit of the memory checker, i.e., $z_i = y_i \vee 1[i \in \pi(x)]$. By completeness, we know that

$$\Pr_{r_1,r_2,z,i}[MC(i, DB_1, \mathsf{st}_1; r_2) = z_i] \geq \frac{99}{100},$$

where we emphasize that DB_1 and st_1 are random variables that depend on r_1 and z. Since the event that $\widetilde{j} = j^*$ is independent of the memory checker by Claim 1, we know

$$\Pr_{r_1,r_2,z,i}\left[MC(i, DB_1, \mathsf{st}_1; r_2) = z_i \,\middle|\, \widetilde{j} = j^*\right] \geq \frac{99}{100}.$$

Given $\widetilde{j} = j^*$, we can invoke Claim 2 and the inclusion-exclusion principle to get

$$\Pr_{r_1,r_2,z,i}\left[MC(i, DB_1, \mathsf{st}_1; r_2) = z_i \text{ and } R\left(i, DB_1, \mathsf{st}_1; r_2\right) \cap \mathrm{BAD} = \emptyset \,\middle|\, \widetilde{j} = j^*\right] \geq \frac{99}{100} + \frac{98}{100} - 1 = \frac{97}{100}.$$

If $R\left(i, DB_1, \mathsf{st}_1; r_2\right) \cap \mathrm{BAD} = \emptyset$, we know that the memory checker using $\widetilde{DB}$ and DB_1 are identical (since they are different only in BAD), so we have

$$\Pr_{r_1,r_2,z,i}\left[MC\left(i, \widetilde{DB}, \mathsf{st}_1; r_2\right) = z_i \text{ and } R\left(i, DB_1, \mathsf{st}_1; r_2\right) \cap \mathrm{BAD} = \emptyset \,\middle|\, \widetilde{j} = j^*\right] \geq \frac{97}{100},$$

which, by dropping the conjunction with the second event, implies

$$\Pr_{r_1,r_2,z,i}\left[MC\left(i, \widetilde{DB}, \mathsf{st}_1; r_2\right) = z_i \,\middle|\, \widetilde{j} = j^*\right] \geq \frac{97}{100}.$$

Then, by an averaging argument[8],

$$\Pr_{r_1,z}\left[\Pr_{i,r_2}\left[MC\left(i,\widetilde{DB},\mathsf{st}_1;r_2\right)=z_i\;\middle|\;r_1,z,\widetilde{j}=j^*\right]>\frac{9}{10}\;\middle|\;\widetilde{j}=j^*\right]\geq\frac{2}{3}.$$

Let G denote these "good" values of (r_1,z) (with density at least $2/3$). By another averaging argument, by restricting to a good value of (r_1,z),

$$\Pr_{i\sim[n]}\left[\Pr_{r_2}\left[MC\left(i,\widetilde{DB},\mathsf{st}_1;r_2\right)=z_i\mid(r_1,z)\in G,i,\widetilde{j}=j^*\right]>\frac{1}{2}\;\middle|\;(r_1,z)\in G,\widetilde{j}=j^*\right]\geq\frac{8}{10}.$$

Since $\tilde{z}_i=\mathsf{BitMajority}\left(MC\left(i,\widetilde{DB},\mathsf{st}_1;r\right)_{r\in\{0,1\}^t}\right)$, this inner event would correspond to success for Bob, so

$$\Pr_{i\sim[n]}\left[\tilde{z}_i=z_i\;\middle|\;(r_1,z)\in G,\widetilde{j}=j^*\right]\geq\frac{8}{10}.$$

Let $I\subseteq[n]$ denote these "good" values of i, where we have $|I|\geq 8n/10$. Assuming $\widetilde{j}=j^*$ and $(r_1,z)\in G$, we notice that this inner probability event corresponds to whether Bob decodes correctly on read i. Assuming $\widetilde{j}=j^*$ and $(r_1,z)\in G$, we therefore know that $\tilde{z}_i=z_i$ is the correct value for all $i\in I$, so Bob can recover a constant fraction of the memory checker's logical bits. That is, there exists $I\subseteq[n]$ with $|I|\geq 8n/10$ such that

$$\left(\widetilde{j}=j^*\wedge(r_1,z)\in G\right)\implies\forall i\in I,\tilde{z}_i=z_i.$$

Now, we argue that with good probability, for all $i\in[n]$, $\tilde{z}_i\in\{z_i,\perp\}$. More precisely:

Claim 3. *There exists a "good" set G' satisfying $\Pr_{r_1,z}[(r_1,z)\in G']\geq 2/3$ and*

$$\left(\widetilde{j}=j^*\wedge(r_1,z)\in G'\right)\implies\forall i\in[n],\tilde{z}_i\in\{z_i,\perp\}.$$

Proof. We invoke soundness of the memory checker on the sequence of logical queries that writes 0 everywhere, writes 1 to $n/10$ random locations denoted by z, and lastly reads to a random location, denoted by i. We will use soundness for the (last) logical read, i.e., query number $n+n/10+1$.

We lastly describe the efficient, dishonest adversary $\mathcal{A}$ used in the soundness game. $\mathcal{A}$ performs a selective replay attack described as follows. For the first $n+n/10$ logical queries, $\mathcal{A}$ behaves exactly honestly, recording all physical operations made by the memory checker so far. Then, for query $n+n/10+1$, the adversary guesses some value $j^*\sim[c]$, which then defines δ_{j^*} and k_{j^*}. Then, for each of the q_r physical accesses corresponding to the read, the adversary uniformly randomly and independently chooses whether to be honest at that location (i.e., whether to use DB_1) or whether to replay that location to DB_0. Note that DB_0 and DB_1 are efficiently computable for the adversary as long as the adversary knows the value k_{j^*}. The guesses here of whether to use DB_0 or DB_1 correspond to a guess for the heavy indices H restricted to the final read.[9] Note that all of the q_r guesses will be correct (i.e., correspond to H) with probability (at least) $1/2^{q_r}$. (We note that this adversary runs in time $\mathsf{poly}(n)$ even if $m=n^{\omega(1)}$ since the adversary can store all physical queries from the memory checker.) See Figure 5 for explicit details.

[8] Here and later, we use the averaging argument that for all $\epsilon,\epsilon_1,\epsilon_2\in(0,1)$ such that $\epsilon_1\cdot\epsilon_2\geq\epsilon$, if $\Pr_{X,Y}[f(X,Y)=1]\geq 1-\epsilon$, then $\Pr_X[\Pr_Y[f(X,Y)=1|X]>1-\epsilon_1]\geq 1-\epsilon_2$.

[9] Note that for a memory checker with secret local state, there is no clear way for the adversary to compute a valid heavy set H with reasonable probability, as H could, for example, be specified by a PRF seed in the memory checker's secret state.

Description of $\mathcal{A}$ (for soundness)

- For the first $n + n/10$ queries (i.e., for the logical writes):
  - Behave honestly, and record all physical queries made by the memory checker.
- Let $DB_1 \in (\{0,1\}^w \cup \{\perp\})^m$ denote the updated (honest) database after the $n + n/10$ queries.
- For query $n + n/10 + 1$ (i.e., for the logical read):
  - Sample $j^* \sim [c]$ uniformly at random, and let $k := k_{j^*}$.
  - Let $DB_0 \in (\{0,1\}^w \cup \{\perp\})^m$ be the database rewound to the end of the first $n + n/10 - k$ logical queries.
  - Generate $\widehat{DB} \in (\{0,1\}^w \cup \{\perp\})^m$ so that for all $v \in [m]$, $\widehat{DB}[v] = DB_0[v]$ with probability 1/2 and $\widehat{DB}[v] = DB_1[v]$ with probability 1/2, independently over all $v \in [m]$.[a]
  - Perform all physical queries from the memory checker with respect to $\widehat{DB}$.

[a] Even if $m = n^{\omega(1)}$, the adversary can lazily generate $DB_0, DB_1, \widehat{DB}$ on the fly by storing all physical queries from the memory checker.

Figure 5: Description of memory checking adversary (for soundness).

Let $\widehat{DB} \in (\{0,1\}^w \cup \{\perp\})^m$ be the random variable corresponding to the uniformly random hybrid database between DB_0 and DB_1 used by the adversary in the replay attack. Soundness for this adversary means

$$\Pr_{\widehat{DB}, r_1, r_2, z, i}\left[MC\left(i, \widehat{DB}, \mathsf{st}_1; r_2\right) = 1 - z_i\right] \leq \frac{1}{10000 \cdot n \cdot q_r \cdot 2^{q_r}}.$$

Using the probability bound $\Pr[A \mid B] \leq \Pr[A]/\Pr[B]$, and letting B denote the event that

$$\widetilde{DB}|_{R(i,\widetilde{DB},\mathsf{st}_1;r_2)} = \widehat{DB}|_{R(i,\widetilde{DB},\mathsf{st}_1;r_2)},$$

which occurs with probability (at least) $1/2^{q_r}$, we have

$$\begin{aligned}\Pr_{r_1, r_2, z, i}\left[MC\left(i, \widehat{DB}, \mathsf{st}_1; r_2\right) = 1 - z_i \;\middle|\; B\right] &\leq \frac{1/(10000 \cdot n \cdot q_r \cdot 2^{q_r})}{1/(2^{q_r})} \\ &= \frac{1}{10000 n q_r}.\end{aligned}$$

Since j^* is sampled by $\mathcal{A}$ after $\widetilde{j}$ is defined, we know that $\widetilde{j} = j^*$ with probability $1/c$. Therefore, we similarly have

$$\begin{aligned}\Pr_{r_1, r_2, z, i}\left[MC\left(i, \widehat{DB}, \mathsf{st}_1; r_2\right) = 1 - z_i \;\middle|\; B, \widetilde{j} = j^*\right] &\leq \frac{1/(10000 n q_r)}{1/c} \\ &= \frac{1}{100n}.\end{aligned}$$

Note that the memory checker using $\widehat{DB}$ and conditioning on B is identical to the memory checker using $\widetilde{DB}$, so we have

$$\Pr_{r_1,r_2,z,i}\left[MC\left(i,\widetilde{DB},\mathsf{st}_1;r_2\right)=1-z_i\;\middle|\;\widetilde{j}=j^*\right]\le\frac{1}{100n}.$$

By negating and similar averaging as before, we have

$$\Pr_{r_1,z}\left[\Pr_{i,r_2}\left[MC\left(i,\widetilde{DB},\mathsf{st}_1;r_2\right)\in\{z_i,\bot\}\;\middle|\;r_1,z,\widetilde{j}=j^*\right]\ge 1-\frac{3}{100n}\;\middle|\;\widetilde{j}=j^*\right]\ge\frac{2}{3}.$$

Let G' denote the set of these "good" values of (r_1,z) (with density at least 2/3). By averaging again and considering $(r_1,z)\in G'$,

$$\Pr_{i\sim[n]}\left[\Pr_{r_2}\left[MC\left(i,\widetilde{DB},\mathsf{st}_1;r_2\right)\in\{z_i,\bot\}\;\middle|\;(r_1,z)\in G',i,\widetilde{j}=j^*\right]>\frac{1}{2}\;\middle|\;(r_1,z)\in G',\widetilde{j}=j^*\right]\ge 1-\frac{1}{10n}.$$

Since $1-1/(10n)>1-1/n$, the inner event holds with probability 1, i.e., for *all* $i\in[n]$. Moreover, since $\tilde{z}_i=\mathsf{BitMajority}\left(MC\left(i,\widetilde{DB},\mathsf{st}_1;r\right)_{r\in\{0,1\}^t}\right)$, we have

$$\left(\widetilde{j}=j^*\wedge(r_1,z)\in G'\right)\implies\forall i\in[n],\tilde{z}_i\in\{z_i,\bot\},$$

as desired. □

That is, assuming $\widetilde{j}=j^*$ and $(r_1,z)\in G'$, then Bob does not decode "incorrectly" for any $i\in[n]$.

In summary, assuming throughout that $\widetilde{j}=j^*$ and $(r_1,z)\in G\cap G'$, none of $\tilde{z}_i$ will be the incorrect bit, and moreover, for $8n/10$ values of $i\in[n]$, $\tilde{z}_i$ will be the correct bit. In particular, Bob can use the map π to pass these values to x, where Bob now only needs to learn which bits of the remaining at most $n-8n/10=n/5$ placeholder values of x are 1.

We now specify Alice's auxiliary string. Alice can run Bob's whole strategy and deduce where Bob will put placeholder values. Alice will simply send some compressed representation of these placeholder values. Explicitly, Alice must send at most $n/5$ bits of z in the worst case. This will be at most

$$\left\lceil\log\left(\binom{n/5}{j}\right)\right\rceil$$

bits for some $j\le k$, where j corresponds to the number of 1s in the $n/5$ placeholder bits. Since $k\le n/100$, this quantity is maximized when $j=k$. As a result, we have the bound

$$|\mathsf{aux}|\le\left\lceil\log\left(\binom{n/5}{k}\right)\right\rceil.$$

Moreover, notice that by definition of G and G' and the union bound, the probability that $\widetilde{j}=j^*$ and $(r_1,z)\in G\cap G'$ is at least $1/c\cdot 1/3=1/(300q_r)$. Therefore, the success probability of this protocol is at least $1/(300q_r)$.

We are finally ready to invoke the compression lemma (Lemma 2). By the information that Alice is communicating to Bob, we have

$$\begin{aligned}
&\log\left(\prod_{j\in[c],j\neq j^*}\binom{9n/10+k_j}{k_j}\right)+p+|\mathsf{aux}|+|H|(w+\log m+2)+1\\
&\geq \log\left(\prod_{j\in[c]}\binom{9n/10+k_j}{k_j}\right)-\log(300q_r).
\end{aligned}$$

This simplifies to

$$p+|\mathsf{aux}|+|H|(w+\log m+2)\geq \log\left(\binom{9n/10+k}{k}\right)-\log(300q_r)-1,$$

which implies, by using the standard bounds on Binomial coefficients $(n/k)^k\leq\binom{n}{k}\leq(en/k)^k$, that

$$\begin{aligned}
p&\geq \log\left(\binom{9n/10}{k}\right)-\log\left(\binom{n/5}{k}\right)-|H|(w+\log m+2)-\log(300q_r)-2\\
&\geq k\log(9n/(10k))-k\log(en/(5k))-|H|(w+\log m+2)-\log(300q_r)-2\\
&=k\log(45/(10e))-|H|(w+\log m+2)-\log(300q_r)-2.
\end{aligned}$$

By further simplifying, we get that

$$\begin{aligned}
p&\geq \frac{k}{2}-|H|(w+\log m+2)-\log(300q_r)-2\\
&\geq \frac{1}{200q_rq_w\delta_{j^*}}-|H|(w+\log m+2)-\log(300q_r)-\frac{5}{2}\\
&=\frac{1}{200q_rq_w}\left(\frac{1}{\delta_{j^*}}-|H|\cdot\underbrace{200q_rq_w(w+\log m+2)}_{\gamma}\right)-\log(300q_r)-\frac{5}{2}.
\end{aligned}$$

Finally, using the guarantee from Lemma 3, we get that

$$\begin{aligned}
p&>\frac{1}{200q_rq_w}\cdot\frac{n}{(400q_rq_w(w+\log m+2))^c}-\log(300q_r)-\frac{5}{2}\\
&\geq\frac{n}{(400q_rq_w(w+\log m+2))^{100q_r+1}}-\log(q_r)-11,\\
&\geq\frac{n}{(600q_rq_w(w+\log m))^{202q_r}}-\log(q_r)-11,
\end{aligned}$$

as desired.

5 Lower Bound for Low Write Complexity

In this section we prove a "write-optimized" lower bound that complements the bound in Theorem 5. However, as opposed to Theorem 5, our lower bound here only applies to deterministic and non-adaptive memory checkers (whereas the lower bound in Theorem 5 had no restriction on the scheme).

Theorem 6. *Consider a* deterministic and non-adaptive *memory checker for a logical memory of size n and with logical word size $w_\ell = 1$, supporting $2n$ logical queries. Assume that the physical database is of size m and with physical word size w, the read query complexity is q_r, the write query complexity is q_w, and the local space is p. If completeness is $99/100$ and soundness is $1/(100n)$, then there exists a universal constant $C \leq 100$ such that for $n \geq C$,*

$$p \geq \frac{n}{(Cq_r q_w w)^{C \cdot q_w}} - C.$$

We obtain (analogously to Corollary 2) the following corollary stating a lower bound on the read query complexity in the case of low write complexity.

Corollary 3. *In the setting of Theorem 6 and further assuming that $w \leq \mathsf{polylog}(n)$ and $p \leq n^{1-\epsilon}$ for some $\epsilon > 0$, the following hold:*

- *If $q_w = o(\log n / \log\log n)$, then $q_r = (\log n)^{\omega(1)}$.*
- *If $q_w = O(1)$, then $q_r = n^{\Omega(1)}$.*

The rest of this section is devoted to the proof of Theorem 6.

Proof of Theorem 6. Throughout, we assume $n \geq 100$ and n is a multiple of 10 for simplicity, but our proof can be easily modified to handle arbitrary $n \geq 100$.

Let $m \leq 2^{\mathsf{poly}(n)}$ be the physical database size. Since the scheme is deterministic and non-adaptive, there exist fixed sequences $W_1, \cdots, W_n \in [m]^{q_w}$ (and $R_1, \cdots, R_n \in [m]^{q_r}$) such that each logical write (and read, respectively) instruction to index $i \in [n]$ performs physical queries exactly according to the sequence W_i (and R_i, respectively), regardless of the local state, contents on the database, or random coins. Without loss of generality, treating W_i and R_i as sets, we assume that each W_i and R_i have exactly q_w and q_r distinct elements, respectively. (If not, we can pad with distinct dummy queries, and our argument still goes through.) Let $\mathcal{D}$ be the distribution defined as follows:

1. Sample $i \sim [n]$ uniformly at random;
2. Select and output a uniformly random element of W_i.

We apply Lemma 3 to $\mathcal{D}$ with $c = 10q_w$ and $\gamma = 200q_r q_w(w+1)$ to get $[m] = L \sqcup M \sqcup H$.

At a high level, our goal will be to restrict the logical indices $[n]$ to those that avoid M and also avoid many logical reads (outside of H). Then, we write to a random subset of these indices to communicate information from Alice to Bob.

Since $\Pr_{X\sim\mathcal{D}}[X \in M] \leq 1/(10q_w)$, we know that $\Pr_{i\sim[n]}[W_i \cap M \neq \emptyset] \leq 1/10$. As a result, there exists $I_0 \subseteq [n]$ of size $|I_0| \geq 9n/10$ such that for all $i \in I_0$, $W_i \cap M = \emptyset$. Furthermore, $\Pr_{X\sim\mathcal{D}}[X = \ell] \leq \delta$ for all $\ell \in L$. Consequently, $\Pr_{i\sim[n]}[\ell \in W_i] \leq \delta q_w$ for all $\ell \in L$. This means that for all $\ell \in L$, there are most $\delta q_w n$ values of $i \in [n]$ such that $\ell \in W_i$, i.e.,

$$\forall \ell \in L : \sum_{i\in[n]} \mathbb{1}[\ell \in W_i] \leq \delta q_w n.$$

For $i \in [n]$, let α_i denote the number of read indices $j \in [n]$ such that $W_i \cap R_j \cap L \neq \emptyset$. By the above argument and counting, we have

$$\begin{aligned}\sum_{i\in[n]} \alpha_i = \sum_{i\in[n]}\sum_{j\in[n]} \mathbb{1}[W_i \cap R_j \cap L \neq \emptyset] &\leq \sum_{i\in[n]}\sum_{j\in[n]}\sum_{\ell\in R_j\cap L} \mathbb{1}[\ell \in W_i] \\ &\leq \sum_{j\in[n]}\sum_{\ell\in R_j\cap L} \delta q_w n \\ &\leq \delta q_w q_r n^2.\end{aligned}$$

Therefore, by Markov's inequality,

$$\Pr_{i\sim[n]}[\alpha_i \geq 10\delta q_w q_r n] \leq \frac{\mathbb{E}_{i\sim[n]}[\alpha_i]}{10\delta q_w q_r n} \leq \frac{1}{10}.$$

Letting $I_1 = \{i \in [n] : \alpha_i < 10\delta q_w q_r n\}$, we have $|I_1| \geq 9n/10$. By the inclusion exclusion principle, it follows that $|I_0 \cap I_1| \geq 8n/10$. For the remainder of the protocol, we fix some arbitrary $I \subseteq I_0 \cap I_1$ such that $|I| = 8n/10$. Note that for all $i \in I$, we have $W_i \cap M = \emptyset$ and the number of read indices $j \in [n]$ such that $W_i \cap R_j \cap L \neq \emptyset$ is at most $10\delta q_w q_r n$.

Protocol description. Now, we construct a public-coin protocol for Alice and Bob to send a uniformly random string $x \in \{0,1\}^{8n/10}$ subject to $\|x\|_0 \leq k$ for $k = \lfloor 1/(100 q_r q_w \delta)\rfloor$. Note that $k \leq n/100$ since $\delta \geq 1/n$. Before Alice sees x, Alice and Bob together (using the same randomness) run a memory checker and write 0 to all locations $i \in [n]$. This gives Alice and Bob the public database DB_0 after these writes as well as the local state st_0. (This can all be formalized by Alice and Bob sharing a sufficiently long uniformly random string and running the memory checker on that string.) Let $\pi : [8n/10] \to [n]$ correspond to some fixed ordering of I, i.e., π bijectively maps $[8n/10]$ to $I \subseteq [n]$.

Alice's encoding. In short, Alice will write x to the memory checker and send some information related to the memory checker at the end of the writes.

More precisely, Alice tosses coins and uses the memory checker (with DB_0 and st_0) to write 1 to every logical index in $\pi(x) := \{\pi(i) : i \in [8n/10], x_i = 1\}$. This produces a new updated public database DB_1 and local state st_1. Alice then sends the following to Bob:

- The updated local state st_1,
- $DB_1|_H$, i.e., the contents of DB_1 at the locations in H, and
- Some auxiliary string aux (specified later) which will help Bob complete Alice's information into a full description of x.

See Figure 6 for explicit details. The total size of this message (in bits) can be upper bounded by $p + |H|(w+1) + |\mathsf{aux}|$.[10]

[10] Footnote 7 applies here as well.

Alice's Strategy

Shared by Alice and Bob: DB_0, st_0
Alice's Private Input: x

1. Using DB_0 and st_0, use the memory checker to write "1" to all indices in $\pi(x) \subseteq [n]$. This results in updated DB_1 and st_1.

2. Run Steps 1-3 of Bob's strategy (Figure 7) to generate the subset $U \subseteq [n]$.

3. Let aux be an encoding of the subset $U \cap \pi(x) \subseteq U$.

Output: $(DB_1|_H, \mathsf{st}_1, \mathsf{aux})$.

Figure 6: Alice's Encoding Strategy.

Bob's decoding. Given $DB_1|_H$ from Alice, Bob can recreate some partially updated public database $\widetilde{DB}$, which is defined as DB_1 on locations in H and DB_0 on $\overline{H} = L \sqcup M$. In short, Bob's strategy will simulate the memory checker on the next logical read to all possible $i \in [n]$, rewinding back each time, using $\widetilde{DB}$ and st_1 from Alice. More formally, Bob will brute force over all $i \in [n]$ and random bits used by the memory checker for this read, and Bob will deduce a bit value $\tilde{z}_i$ for z_i, the ith logical bit of the memory checker if there is a strict majority of choices of randomness that agree on a bit (not $\perp$) to output. Otherwise, Bob will put some placeholder value for the ith bit and use aux from Alice to fill it in.

Explicitly, let $r \in \{0,1\}^t$ denote the random bits used by the memory checker (for this final read). Let $MC(i, DB, \mathsf{st}; r) \in \{0, 1, \perp\}$ denote the output of the memory checker upon performing a logical read to index $i \in [n]$ from local state st and public database DB using randomness r. For each $i \in [n]$, Bob will set

$$\tilde{z}_i := \mathsf{BitMajority}\left(MC\left(i, \widetilde{DB}, \mathsf{st}_1; r\right)_{r \in \{0,1\}^t} \right).$$

Reading off the values in I will then yield Bob's output $\tilde{x} \in \{0,1\}^{8n/10}$ by the mapping π. See Figure 7 for explicit details.

We now analyze how successful Bob will be. Recall that DB_0 and DB_1 can differ only at locations that Alice accessed in writing 1 to $\pi(x)$. Let

$$W := \bigcup_{i \in \pi(x)} W_i$$

denote this set. Since $\|x\|_0 = k$, we have $|W| \le kq_w$. Define $\mathrm{BAD} := W \cap (L \cup M) = W \cap \overline{H}$ to be the set of locations that Alice wrote to that are not included in H. Equivalently, DB_1 and $\widetilde{DB}$ differ only at physical locations in BAD. (We emphasize that Bob does not know the set BAD.) Moreover, since $\pi(x) \subseteq I$, by construction of I, we know that $W \cap M = \emptyset$. Therefore, $\mathrm{BAD} = W \cap L$.

We now determine how many read indices $j \in [n]$ are unaffected by these writes, in the sense of $R_j \cap \mathrm{BAD} = \emptyset$. Note that if $R_j \cap \mathrm{BAD} = \emptyset$, then Bob using $\widetilde{DB}$ to perform logical read j

Bob's Strategy

Shared by Alice and Bob: DB_0, st_0
Alice's Message: $(DB_1|_H, \mathsf{st}_1, \mathsf{aux})$.

1. Using DB_0 from the shared input and $DB_1|_H$ from Alice, define the database $\widetilde{DB} \in (\{0,1\}^w \cup \{\bot\})^m$ as follows:

$$\widetilde{DB}[v] := \begin{cases} DB_1[v] & \text{if } v \in H, \\ DB_0[v] & \text{otherwise.} \end{cases}$$

2. For all $j \in [n]$, let

$$\tilde{z}_j := \mathsf{BitMajority}\left(MC\left(j, \widetilde{DB}, \mathsf{st}_1; r\right)_{r \in \{0,1\}^t} \right) \in \{0, 1, \bot\}.$$

3. Define $U := \{j \in [n] : \tilde{z}_j = \bot\}$.

4. Parsing $\mathsf{aux} \subseteq U$, define $z \in \{0,1\}^n$ by

$$z_j := \begin{cases} \tilde{z}_j & \text{if } \tilde{z}_j \neq \bot, \\ 1 & \text{if } \tilde{z}_j = \bot \text{ and } j \in \mathsf{aux}, \\ 0 & \text{if } \tilde{z}_j = \bot \text{ and } j \notin \mathsf{aux}. \end{cases}$$

5. Define $x \in \{0,1\}^{8n/10}$ by $x_\ell := z_{\pi(\ell)} \in \{0,1\}$.

Output: x.

Figure 7: Bob's Decoding Strategy.

is identical to instead using DB_1 to perform logical read j honestly, and hence, we can invoke completeness. Let $J = \{j \in [n] : R_j \cap \mathrm{BAD} = \emptyset\}$. By construction of I, each $i \in \pi(x)$ satisfies $r_i < 10\delta q_w q_r n$, which implies

$$\begin{aligned} \left|\overline{J}\right| = |\{j \in [n] : R_j \cap W \cap L \neq \emptyset\}| &\leq \sum_{i \in \pi(x)} |\{j \in [n] : R_j \cap W_i \cap L \neq \emptyset\}| \\ &= \sum_{i \in \pi(x)} \alpha_i < 10k\delta q_w q_r n \leq \frac{n}{10}, \end{aligned}$$

where the last inequality follows from the definition of k. Therefore, $|J| \geq 9n/10$. We have shown the following claim:

Claim 4. *There exists $J \subseteq [n]$ such that $|J| \geq 9n/10$ and for all $j \in J$, $R_j \cap \mathrm{BAD} = \emptyset$, i.e.,*

$$DB_1\big|_{\bigcup_{j \in J} R_j} = \widetilde{DB}\big|_{\bigcup_{j \in J} R_j}. \tag{2}$$

Now, we invoke completeness of the memory checker. Explicitly, we invoke completeness on the sequence of logical instructions that writes 0 everywhere, writes 1 to $\pi(I)$, and lastly reads to some uniformly random $j \sim J$. We will invoke completeness for logical instruction $n+k+1$, namely, for the read. Letting r_1, r_2, and j denote the randomness of the memory checker used for all of the logical writes, the randomness of the memory checker used for the final logical read, and the index of the final logical read, respectively, we know by completeness that

$$\Pr_{r_1,r_2,j}[MC(j, DB_1, \mathsf{st}_1; r_2) = z_j] \geq \frac{99}{100}.$$

By (2), since $j \in J$, we have

$$\Pr_{r_1,r_2,j}\left[MC\left(j, \widetilde{DB}, \mathsf{st}_1; r_2\right) = z_j\right] \geq \frac{99}{100}.$$

By averaging, we get that

$$\Pr_{r_1}\left[\Pr_{r_2,j}\left[MC\left(j, \widetilde{DB}, \mathsf{st}_1; r_2\right) = z_j\right] > \frac{29}{30} \,\middle|\, r_1\right] \geq \frac{2}{3}.$$

Let G denote these set of "good" r_1 values with density at least 2/3. By averaging again and conditioning on $r_1 \in G$, we have

$$\Pr_{j\sim J}\left[\Pr_{r_2}\left[MC\left(j, \widetilde{DB}, \mathsf{st}_1; r_2\right) = z_j \mid r_1 \in G\right] > \frac{1}{2} \,\middle|\, r_1 \in G\right] \geq \frac{9}{10}.$$

Note that assuming $r_1 \in G$, this inner probability event corresponds to whether Bob's jth decoding will be correct. That is,

$$\Pr_{j\sim J}\left[\tilde{z}_i = z_i \,\middle|\, r_1 \in G\right] \geq \frac{9}{10}.$$

Let $J' \subseteq J$ denote this set of "good" j values, where we have $|J'| \geq 9|J|/10 \geq 8n/10$. By construction, if $r_1 \in G$, we have $\tilde{z}_j = z_j$ for all $j \in J'$. To summarize, there exists $J' \subseteq [n]$ with $|J'| \geq 8n/10$ and G with density at least 2/3 such that

$$r_1 \in G \implies \forall j \in J', \tilde{z}_j = z_j.$$

Next, we argue that with good probability, for all $j \in [n]$, $\tilde{z}_j \in \{z_j, \perp\}$. More precisely:

Claim 5. *There exists a "good" set G' satisfying $\Pr_{r_1}[r_1 \in G'] \geq 2/3$ and*

$$r_1 \in G' \implies \forall j \in [n], \tilde{z}_j \in \{z_j, \perp\}.$$

Proof. We invoke soundness of the memory checker on the sequence of logical instructions that writes 0 everywhere honestly, writes 1 to all of $\pi(x)$, and then performs a random read to $j \sim [n]$. We invoke soundness on query $n+k+1$, i.e., the logical read.

We apply soundness with the following efficient, dishonest adversary, doing a selective replay attack described as follows. The adversary will perform a replay attack to DB_0 on all locations outside of H, and the adversary will be honest on all locations in H. (Note that Lemma 3 implicitly bounds $|H| \le n$ since $\delta \ge 1/n$, so a non-uniform PPT adversary in n can perform this attack even if $m = n^{\omega(1)}$.) This corresponds to Bob's decoding strategy. Soundness for this adversary means

$$\Pr_{r_1,r_2,j}\left[MC\left(j, \widetilde{DB}, \mathsf{st}_1; r_2\right) = 1 - z_j\right] \le \frac{1}{100n}.$$

By similar averaging as before,

$$\Pr_{r_1}\left[\Pr_{r_2,j}\left[MC\left(j, \widetilde{DB}, \mathsf{st}_1; r_2\right) = 1 - z_j\right] \le \frac{1}{30n} \,\middle|\, r_1\right] \ge \frac{2}{3}.$$

Let G' denote these set of "good" r_1 values, with density at least $2/3$. By averaging again and conditioning on $r_1 \in G'$, we have

$$\Pr_{j\sim[n]}\left[\Pr_{r_2}\left[MC\left(j, \widetilde{DB}, \mathsf{st}_1; r_2\right) = 1 - z_j \mid r_1 \in G'\right] < \frac{1}{2} \,\middle|\, r_1 \in G'\right] > 1 - \frac{1}{n}.$$

Therefore, the inner event holds for all $j \in [n]$. As such,

$$r_1 \in G' \implies \forall j \in [n], \tilde{z}_j \in \{z_j, \perp\}$$

□

In summary, assuming that $r_1 \in G \cap G'$, none of z_i will be incorrect, and moreover, for $8n/10$ values of $i \in [n]$, z_i will be correct. In particular, Bob can use the map π to pass these values to x, where Bob now only needs to learn which bits of the remaining at most $n/5$ placeholder values of x are 1. We can use aux to specify these placeholder values, giving

$$|\mathsf{aux}| \le \left\lceil \log\left(\binom{n/5}{k}\right)\right\rceil.$$

By the inclusion-exclusion principle, we know $\Pr[r_1 \in G \cap G'] \ge 1/3$, and as such, the success probability of this protocol is at least $1/3$. We can therefore invoke Lemma 2 to get

$$p + |H|(w+1) + |\mathsf{aux}| \ge \log\left(\binom{8n/10}{k}\right) - \log(3).$$

By standard Binomial coefficient bounds, we have

$$\begin{aligned} p \ge k\log(4/e) - |H|(w+1) - \log(3) - 1 &\ge \frac{1}{200\delta q_r q_w} - |H|(w+1) - \log(3) - \frac{3}{2} \\ &= \frac{1}{200 q_r q_w}\left(\frac{1}{\delta} - |H| \cdot \underbrace{200 q_r q_w (w+1)}_{\gamma}\right) - \log(3) - \frac{3}{2}. \end{aligned}$$

Finally, using the guarantee from Lemma 3, we get that

$$\begin{aligned} p &\ge \frac{1}{200 q_r q_w} \cdot \frac{n}{(400 q_r q_w (w+1))^{10 q_w}} - \log(3) - \frac{3}{2} \\ &\ge \frac{n}{(400 q_r q_w (w+1))^{11 q_w}} - 3 \ge \frac{n}{(20 w q_r q_w)^{50 q_w}} - 3, \end{aligned}$$

as desired. □

6 Optimal Memory Checker Construction

We give a new construction of a memory checker with the following features: (1) it is deterministic and non-adaptive (see Definition 6), (2) its query complexity is $q = \Theta(\log n/\log\log n)$ (which is optimal up to constants due to our lower bound), and (3) it satisfies perfect completeness and s-strong-soundness for s negligible, assuming the existence of (sub-exponentially secure) one-way functions. Recall that strong soundness (see Definition 5) in particular allows the adversary to view the local state and random tape of the memory checker when adaptively selecting queries. Our construction can support logical words of size $w_\ell = \Theta(w)$, resulting in a bandwidth of $\Theta(\log n/\log\log n)$.

In comparison, prior works did not satisfy all of the above properties simultaneously. Most relevantly, the construction of [BEG$^+$94] had query complexity $q = \Theta(\log n)$ and the construction of [PT11] assumed that the local state of the user is private from the adversary (allowing them to use PRFs).

Overview. We now give a brief overview of the construction. We refer to Figure 8 for an illustration. We use a tree-based construction, combining elements of previous constructions [BEG$^+$94, PT11]. Blum et al. [BEG$^+$94] give a binary UOWHF tree construction of a memory checker with *public* local state and $O(\log n)$ query complexity. On the other hand, Papamanthou and Tamassia [PT11] give a PRF tree construction of a memory checker with *secret* local state (to store the PRF key) and $O(\log n/\log\log n)$ query complexity. Our construction gets the best of both worlds: a memory checker with public local state and $O(\log n/\log\log n)$ query complexity.

Our main idea is as follows: We will use a d-ary UOWHF tree for some $d = \mathsf{polylog}(n)$, making the depth of the tree $\Theta(\log_d n) = \Theta(\log n/\log\log n)$. The direct construction of Merkle/UOWHF trees (as in [BEG$^+$94]) would require reads and writes to look at all siblings, causing the read and write query complexities to be $O(d\log_d n)$. However, we can use the following trick to make the query complexities $O(\log_d n)$: by using physical word size $w = \mathsf{polylog}(n)$, each node can store the hashes of all of its children nodes. Then, to recompute hashes for reads and writes, we only need 1 query per level instead of d. This can also be seen as copying the contents of the UOWHF tree "one layer up" so that sibling nodes do not need to be queried.

We remark that the idea of using a larger physical word size (though still $\mathsf{polylog}(n)$) to "pack" all d siblings into a single word was also used by Papamanthou and Tamassia [PT11], but there the context is slightly different because they rely on a tree of pseudorandom values and assume that the local state can be used to store a secret PRF key.

Theorem 7. *Suppose there exist $(T(\lambda), \epsilon(\lambda))$-secure UOWHFs. Then, for $\lambda \geq 2\sqrt{\log n}$, there is a deterministic and non-adaptive memory checker with public local state for a logical memory of size n that has query complexity $O(\log n/\log\log n)$, local space $O(\lambda)$, logical and physical word size $\Theta(\lambda\sqrt{\log(n)})$, completeness 1, soundness $\epsilon(\lambda)\cdot\mathsf{poly}(n)$, and public database size $O(n\sqrt{\log n})$ that is secure against $T(\lambda) - \mathsf{poly}(n, \lambda)$ time adversaries.*

Proof. We use a d-ary tree for $d = \lfloor\sqrt{\log(n)}\rfloor$ with $\ell = \lceil\log_d(n)\rceil + 1$ levels, so that there are at least n leaves. We identify the vertices $v \in V$ in this tree by the set

$$V = [d]^{<\ell} = \bigcup_{0\leq i<\ell} [d]^i,$$

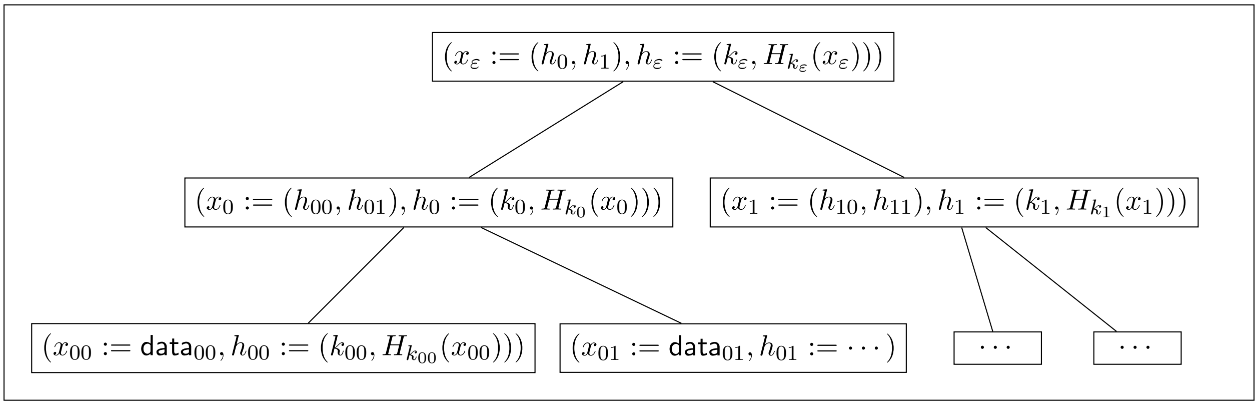
$(x_\varepsilon := (h_0, h_1), h_\varepsilon := (k_\varepsilon, H_{k_\varepsilon}(x_\varepsilon)))$
$(x_0 := (h_{00}, h_{01}), h_0 := (k_0, H_{k_0}(x_0)))$
$(x_1 := (h_{10}, h_{11}), h_1 := (k_1, H_{k_1}(x_1)))$
$(x_{00} := \mathsf{data}_{00}, h_{00} := (k_{00}, H_{k_{00}}(x_{00})))$
$(x_{01} := \mathsf{data}_{01}, h_{01} := \cdots)$
$\cdots$
$\cdots$

Figure 8: This tree represents the construction of the memory checker in Theorem 7 with arity $d = 2$, $n = 4$ logical elements, and $\ell = \lceil \log_d(n) \rceil + 1 = 3$ levels. The four logical elements are $\mathsf{data}_{00}, \mathsf{data}_{01} \in \{0,1\}^{2d\lambda}$ (shown in the figure) and $\mathsf{data}_{10}, \mathsf{data}_{11} \in \{0,1\}^{2d\lambda}$ (not shown). Note that to recompute and check hashes in the tree, there is no need to read any siblings, so the query complexity for updating and checking the hashing is $O(\log_d n)$.

where we use the notation ε to denote the empty string, i.e., the unique element of $[d]^0$. We associate each vertex in this tree with a physical location, making the public database have $\sum_{i=0}^{\ell-1} d^i \leq O(dn) = O(n\sqrt{\log n})$ many physical words. We associate each leaf of this tree (of which there are at least n) with a logical index. The logical word size will be $w_\ell = 2d\lambda = \Theta(\lambda\sqrt{\log n})$, and the physical word size will be $w = (2d+2)\lambda = \Theta(\lambda\sqrt{\log n})$. For a leaf vertex $v \in [d]^{\ell-1}$, let $\mathsf{data}_v \in \{0,1\}^{w_\ell}$ denote the logical data for the memory checker at the corresponding logical index.

For a vertex $v \in V$, the corresponding entry in the public database will consist of (x_v, h_v) pairs, where $|x_v| = 2d\lambda$ and $|h_v| = 2\lambda$, defined recursively as follows:

$$x_v = \begin{cases} \mathsf{data}_v & \text{if } v \in [d]^{\ell-1} \text{ (i.e., if } v \text{ is a leaf)}, \\ (h_{v||1}, h_{v||2}, \cdots, h_{v||d}) & \text{otherwise}, \end{cases}$$
$$h_v = (k_v, H_{k_v}(x_v)) \,.$$

Here and throughout, H_k denotes an element of the UOWHF function family with key k. Note that the size of the input that we are feeding into the hash function is $2d\lambda \leq 2\lambda\sqrt{\log n}$, so using this hash function (which has domain $\{0,1\}^{\lambda^2}$ by Definition 8) is well-defined as long as $\lambda \geq 2\sqrt{\log n}$. The local state of the memory checker will simply consist of $h_\varepsilon = (k_\varepsilon, H_{k_\varepsilon}(x_\varepsilon))$, which has size 2λ.

Handling reads. We now describe how logical reads are performed. To read index $i \in [n]$, the memory checker will first read the leaf v corresponding to logical index i to yield a candidate response $x_v = \mathsf{data}_v \in \{0,1\}^{w_\ell}$.

Next, the memory checker performs reads up the path from v to the root ε to verify the validity of data_v, starting at node $u = v$ and going to $u = \varepsilon$. Specifically, at each u, the memory checker computes $H_{k_u}(x_u)$ and checks that it is consistent with h_u, and additionally, for internal nodes, checks that x_u contains the correct value $h_{u||j}$ for the unique child j it has read. Finally, the memory checker also checks whether the value of h_ε it has read from the public database is consistent with its local state, which has an uncorrupted copy of h_ε. If all checks go through, the memory checker returns data_v to the user; otherwise, the memory checker returns $\bot$.

Overall, the query complexity of a read is $\ell = \Theta(\log_d n) = \Theta(\log n / \log\log n)$.

Handling writes. We now describe how logical writes are performed. To write data to logical index $i \in [n]$, let v be the corresponding leaf to index i. For each node up the path from v to the root ε, the memory checker will perform one physical read and then one physical write before proceeding to the next level.

For the physical write to leaf v, the memory checker will sample fresh $k_v \sim \{0,1\}^\lambda$ and then write $x_v = \mathsf{data}$ and $h_v = (k_v, H_{k_v}(x_v))$. For the physical write to internal nodes u on the path from v to ε, the memory checker samples fresh $k_u \sim \{0,1\}^\lambda$, updates x_u with its new entry (based on its immediately previous physical read on the same level and physical write one level below), and writes $h_u = (k_u, H_{k_u}(x_u))$. Concurrently, based on values read during the physical reads, the memory checker does all the same checks as a logical read to ensure that all (old) hashes along the way are consistent with its (old) root and (old) local state. Assuming all checks go through, the memory checker updates its local state to the new value of h_ε; otherwise, the memory checker can just store $\bot$ locally and return $\bot$ for any later reads.

Overall, the query complexity of a write is $2\ell = \Theta(\log n / \log\log n)$.

Completeness and soundness. It is clear that this protocol has completeness 1, but it remains to show strong soundness. The analysis proceeds exactly as in Theorem 2 of Blum et al. [BEG$^+$94]. To see strong soundness, suppose not. Then, there is an adversary that breaks strong soundness on one of the reads with probability at least p. Going from the root ε to the leaf for this logical read, there must exist a first node u such that h_u is incorrect. Moreover, u cannot be the root, because h_ε is stored reliably in the local state. Therefore, there must exist a parent v of u, i.e., $u = v||j$ for some $j \in [d]$. Furthermore, by our definition of u, we know that h_v is correct. Thus, k_v and $H_{k_v}(x_v)$ are correct, but x_v, which contains $h_{v||j} = h_u$, must be incorrect because h_u is incorrect. Therefore, this adversary has produced some dishonest $x'_v \neq x_v$ such that $H_{k_v}(x_v) = H_{k_v}(x'_v)$.

We can now use this adversary to created a targeted collision. Specifically, the adversary will guess which read and which node u will be the first incorrect node, and output the x_v as the target corresponding to the honest value before the memory checker has broken soundness. Then, upon receiving $k_v \sim \{0,1\}^\lambda$, the adversary will produce the x'_v that causes a collision. If the adversary breaks memory checking soundness with probability p, then this attack breaks targeted collision resistance with probability $p/\mathsf{poly}(n)$ (due to the guess of the read and node). Therefore, if we have $\epsilon(\lambda)$ security for the UOWHF, then we have soundness at most $\epsilon(\lambda) \cdot \mathsf{poly}(n)$. □

We now apply Theorem 7 with sub-exponentially secure UOWHFs, which we know exist if sub-exponentially secure one-way functions exist [Rom90, KK05].

Corollary 4. *Suppose there exist sub-exponentially secure one-way functions. Then, there is a deterministic and non-adaptive memory checker with public local state for a logical memory of size n that has query complexity $O(\log n / \log\log n)$, local space $\mathsf{polylog}(n)$, equal logical and physical word sizes of $\mathsf{polylog}(n)$, completeness 1, soundness $\mathsf{negl}(n)$, and public database size $O(n\sqrt{\log n})$ that is secure against all $\mathsf{poly}(n)$-time adversaries.*

Remark 6. *We can similarly use the construction in Theorem 7 with larger arity $d \in [\sqrt{\log n}, n]$ to get a memory checker with query complexity $\Theta(\log_d n)$ at the cost of physical and logical words* *of size $d \cdot \mathsf{polylog}(n)$ (and by using UOWHF domain extension). This shows that the dependence on w in Theorem 5 is necessary.*

In particular, for matching physical and logical word size n^{ε}, there is a memory checker with query complexity $\Theta(1/\varepsilon)$. Interestingly, note that this is not *true for ORAM: known lower bounds show that for matching physical and logical word size $n^{\Omega(1)}$, there is still a $\Omega(\log n)$ query complexity lower bound [LN18, KL21].*

Proof of Corollary 4. By assumption, we know there is some $\delta \in (0,1)$ for which we have $(2^{\lambda^{\delta}}, 1/2^{\lambda^{\delta}})$-secure UOWHFs. Setting $\lambda = \log(n)^{2/\delta}$ in Theorem 7 gives the desired result. In fact, soundness is at most (some) quasi-polynomial in n, and it is secure against adversaries with run-time (some) quasi-polynomial in n. (We note that this construction can be optimized to get word sizes $\log(n)^{\delta+\epsilon}$ for any $\epsilon > 0$, so as δ approaches 1, the word size approaches $\log n$.) □

We now give another corollary of this construction where there is a trade-off between local space and query complexity.

Corollary 5. *Suppose there exist sub-exponentially secure one-way functions. Then, for any $q \leq \log n / \log\log n$, there is a deterministic and non-adaptive memory checker with public local state for a logical memory of size n that has query complexity q, local space p, physical word size $\mathsf{polylog}(n)$, completeness 1, soundness $\mathsf{negl}(n)$, and public database size $O(n\sqrt{\log n})$ that is secure against all $\mathsf{poly}(n)$-time adversaries, that satisfies the bound*

$$p \leq \frac{n \cdot \mathsf{polylog}(n)}{(\log n)^{\Omega(q)}}.$$

Proof. We start with the construction of Theorem 7. Recall that the reads and writes have query complexity at most $q' = 2\log_d n + \Theta(1)$. Then, we modify the construction so that the memory checker stores all of the h_v values for layer α of the d-ary tree instead of the root layer, where $\alpha = (q' - q)/2 + \Theta(1)$ is chosen so that the memory checker now needs to only make q queries instead of q' queries. The local space is now

$$p := 2\lambda d^{\alpha} \leq 2\lambda d^{\log_d(n) - q/2 + O(1)} \leq \frac{n \cdot \mathsf{polylog}(n)}{(\log n)^{\Omega(q)}}.$$

□

Acknowledgements

We thank Moni Naor for very useful discussions.

For the third author, research was partially done at NTT Research. His research is further supported in part by DARPA under Agreement No. HR00112020023, NSF CNS-2154149, NSF DGE-2141064, and a Simons Investigator award.

References

[ABC$^+$07] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Xiaodong Song. Provable data possession at untrusted stores. In *CCS*, pages 598–609. ACM, 2007. 1

[AKL$^+$23] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. *J. ACM*, 70(1):4:1–4:70, 2023. 5

[AKLS21] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. Oblivious RAM with worst-case logarithmic overhead. In *CRYPTO*, pages 610–640, 2021. 5

[BCHO22] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. Gemini: Elastic SNARKs for diverse environments. In *EUROCRYPT*, pages 427–457, 2022. 1

[BDFG21] Dan Boneh, Justin Drake, Ben Fisch, and Ariel Gabizon. Halo infinite: Proof-carrying data from additive polynomial commitments. In *CRYPTO*, pages 649–680, 2021. 1

[BEG$^+$94] Manuel Blum, Will Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. *Algorithmica*, 12:225–244, 1994. 1, 3, 5, 12, 37, 39

[BMM$^+$21] Benedikt Bünz, Mary Maller, Pratyush Mishra, Nirvan Tyagi, and Psi Vesely. Proofs for inner pairing products and applications. In *ASIACRYPT*, pages 65–97, 2021. 1

[BN16] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *ITCS*, pages 357–368, 2016. 5

[BR97] Mihir Bellare and Phillip Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. In *CRYPTO*, pages 470–484, 1997. 18

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR Cryptol. ePrint Arch.*, page 86, 2016. 1

[CDH20] David Cash, Andrew Drucker, and Alexander Hoover. A lower bound for one-round oblivious RAM. In *TCC*, pages 457–485, 2020. 2, 5

[CKW17] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious RAM. *J. Cryptol.*, 30(1):22–57, 2017. 1

[CSG$^+$05] Dwaine E. Clarke, G. Edward Suh, Blaise Gassend, Ajay Sudan, Marten van Dijk, and Srinivas Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *IEEE S&P*, pages 139–153, 2005. 1

[DNRV09] Cynthia Dwork, Moni Naor, Guy N Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be? In *TCC*, pages 503–520, 2009. 1, 3, 4, 5, 6, 7, 12, 17

[DTT10] Anindya De, Luca Trevisan, and Madhur Tulsiani. Time space tradeoffs for attacks against one-way functions and PRGs. In *CRYPTO*, pages 649–665, 2010. 12

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, 1986. 17, 18, 20, 45

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, May 1996. 2, 4, 5

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, pages 182–194, 1987. 4

[Gol01] Oded Goldreich. *The Foundations of Cryptography - Volume 1: Basic Techniques.* Cambridge University Press, 2001. 18

[Gol23] Oded Goldreich. On the lower bound on the length of relaxed locally decodable codes. *Electron. Colloquium Comput. Complex.*, TR23-064, 2023. 9, 10, 20

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999. 18, 20, 45

[HJ05] William Eric Hall and Charanjit S. Jutla. Parallelizable authentication trees. In *Selected Areas in Cryptography*, pages 95–109, 2005. 1

[JJ07] Ari Juels and Burton S. Kaliski Jr. PORs: proofs of retrievability for large files. In *CCS*, pages 584–597. ACM, 2007. 1

[KK05] Jonathan Katz and Chiu-Yuen Koo. On constructing universal one-way hash functions from arbitrary one-way functions. *Cryptology ePrint Archive*, 2005. 18, 39

[KL21] Ilan Komargodski and Wei-Kai Lin. A logarithmic lower bound for oblivious RAM (for all parameters). In *CRYPTO*, pages 579–609, 2021. 5, 40

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *CRYPTO*, pages 523–542, 2018. 5, 40

[LPM$^+$13] Jacob R. Lorch, Bryan Parno, James W. Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: ensuring private access to large-scale data in the data center. In *FAST*, pages 199–214. USENIX, 2013. 5

[Mat23] Surya Mathialagan. Memory checking for parallel RAMs. *IACR Cryptol. ePrint Arch.*, page 1703, 2023. Accepted to TCC 2023. 4

[MV23] Surya Mathialagan and Neekon Vafa. Macorama: Optimal oblivious RAM with integrity. In *CRYPTO*, pages 95–127, 2023. 5, 43, 44

[NR09] Moni Naor and Guy N Rothblum. The complexity of online memory checking. *Journal of the ACM*, 56(1):1–46, 2009. 1, 12, 20

[NW93] Noam Nisan and Avi Wigderson. Rounds in communication complexity revisited. *SIAM J. Comput.*, 22(1):211–219, 1993. 2

[NY89] Moni Naor and Moti Yung. Universal one-way hash functions and their cryptographic applications. In *STOC*, pages 33–43, 1989. 17

[OR07] Alina Oprea and Michael K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In *USENIX*, 2007. 1

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, pages 514–523. ACM, 1990. 4

[OWWB20] Alex Ozdemir, Riad S. Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *USENIX*, pages 2075–2092, 2020. 1

[PPRY18] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious RAM with logarithmic overhead. In *FOCS*, pages 871–882, 2018. 5

[PT11] Charalampos Papamanthou and Roberto Tamassia. Optimal and parallel online memory checking. *Cryptology ePrint Archive*, 2011. 2, 3, 37, 47

[RFY$^+$13] Ling Ren, Christopher W. Fletcher, Xiangyao Yu, Marten van Dijk, and Srinivas Devadas. Integrity verification for path oblivious-RAM. In *HPEC*, pages 1–6, 2013. 5

[Rom90] John Rompel. One-way functions are necessary and sufficient for secure signatures. In *STOC*, pages 387–394, 1990. 18, 39

[Set20] Srinath T. V. Setty. Spartan: Efficient and general-purpose zkSNARKs without trusted setup. In *CRYPTO*, pages 704–737, 2020. 1

[SvDS$^+$13] Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *CCS*, pages 299–310, 2013. 5

[SW13] Hovav Shacham and Brent Waters. Compact proofs of retrievability. *J. Cryptol.*, 26(3):442–483, 2013. 1

[WLPZ23] Weijie Wang, Yujie Lu, Charalampos Papamanthou, and Fan Zhang. The locality of memory checking. *IACR Cryptol. ePrint Arch.*, page 1358, 2023. Accepted to CCS 2023. 4

[WTS$^+$18] Riad S. Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zkSNARKs without trusted setup. In *IEEE S&P*, pages 926–943, 2018. 1

[XZZ$^+$19] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *CRYPTO*, pages 733–764, 2019. 1

[ZXZS20] Jiaheng Zhang, Tiancheng Xie, Yupeng Zhang, and Dawn Song. Transparent polynomial delegation and its applications to zero knowledge proof. In *IEEE S&P*, pages 859–876, 2020. 1

A Impossibility of Efficient Malicious Compilers for ORAMs

In this section, we prove that our lower bound also resolves a question left open by Mathialagan and Vafa [MV23] about transforming honest-but-curious Oblivious RAM (ORAM) protocols into maliciously secure ones.

Mathialagan and Vafa [MV23] show that the existence of a black-box RAM program that transforms an honest-but-curious ORAM into a maliciously secure one is *equivalent* to a restricted form of memory checkers, which they call *separated memory checkers* [MV23, Section 5]. In short, a separated memory checker differs from a standard memory checker in two key ways:

1. The main difference is that unlike standard memory checking, where there is a joint PPT adversary $\mathcal{A}$ controlling both the logical queries and the physical database simultaneously, in a separated memory checker, the adversaries controlling the logical queries and the physical database do not collude during the protocol. That is, soundness (and completeness) holds only for "separated" adversaries that control the logical queries and the physical database independently. (The adversaries are allowed to "collude" before starting the protocol.) Our definitions of completeness and soundness (Definitions 3 and 4) satisfy this notion as well, because the sequence of logical queries is fixed in advance of running the experiment.

2. To make their result meaningful for ORAMs, which inherently have small local space, Mathialagan and Vafa [MV23] consider compilers that are only required to work for low-space honest-but-curious ORAM protocols. They show that such compilers imply separated memory checkers that are only required to be sound (and complete) if the adversary controlling the logical queries has a similar space bound. If this space bound for the adversary is p, they call such a separated memory checker a *separated memory checker for users with space p.*

Formally, they show the following:

Theorem 8 ([MV23], Theorem 3)**.** *Suppose there is a RAM program Π such that for all honest-but-curious ORAMs $\mathcal{C}$ with local space at most p, the composition of Π and $\mathcal{C}$ is a maliciously secure ORAM. If Π has blowup in query complexity ℓ, then there is a separated memory checker for users with space $O(p)$ that has query complexity $O(\ell)$ bits.*

Our lower bound in Theorem 5 also rules out separated memory checkers. As a result, we have the following theorem (for the standard setting of memory checking parameters):

Theorem 9. *There does not exist a separated memory checker for a memory of size n with query complexity $o(\log n/\log\log n)$. Moreover, for all $\varepsilon > 0$, the same holds even for separated memory checkers for users with space $O(n^{\varepsilon})$.*

An immediate corollary of this theorem is the following:

Corollary 6. *There does not exist a RAM program Π with blowup in query complexity $o(\log n/\log\log n)$ such that for all honest-but-curious ORAMs $\mathcal{C}$ on a memory of size n, the composition of Π and $\mathcal{C}$ is a maliciously secure ORAM on a memory of size n. The same holds for all $\varepsilon > 0$ even when restricted to honest-but-curious ORAMs with space $O(n^{\varepsilon})$.*

We now prove the main theorem in this section, Theorem 9.

Proof of Theorem 9. The proof proceeds directly from Theorem 5 but considering the memory checker restricted to the first $n' := n^{\varepsilon/2}$ logical indices. The crucial observation is that for this proof, the logical queries can be implemented by an adversary running in space $O(n'\log(n')) = O(n^{\varepsilon/2}\log n) = O(n^{\varepsilon})$ bits. If so, then since $n' = n^{\varepsilon/2}$, Theorem 5 implies a query complexity lower bound of $\Omega(\log(n')/\log\log(n')) = \Omega(\log n/\log\log n)$, as desired.

Recall the sequence of logical instructions: always write 0 to all $i \in [n']$, write 1 to $n'/10$ distinct uniformly random indices in $[n']$ in a uniformly random order, and lastly read to a random index $i \sim [n']$. We claim that this can be implemented in space $O(n' \log(n'))$ bits. The only non-trivial step is writing 1 to $n'/10$ random indices in $[n']$ in a uniformly random order. To do so, the adversary can first generate the set of logical indices to write to as follows. The adversary iterates over $i \in [n']$ and keeps track of a count ctr of the number of 1s written so far. For each i, the adversary will include i in the set independently with probability

$$\frac{n'/10 - \mathsf{ctr}}{n' - i + 1},$$

by sampling a Bernoulli random variable accordingly. After writing down this set, which can be written as a list using $O(n' \log(n'))$ bits, the adversary can uniformly randomly shuffle the list. Finally, the adversary can issue logical writes in that order. □

B Obtaining Polynomial-Size Physical Memory Generically

We show that for memory checkers with a fixed number of supported logical queries, one can assume without loss of generality that $m \leq \mathsf{poly}(n)$:

Lemma 4. *Let $T = T(n)$ be some fixed polynomial. Suppose there is a memory checker with completeness c, soundness s, query complexity $q \leq \mathsf{poly}(n)$, and local space p for a memory of size n that has physical database size m. Suppose this memory checker supports at least T queries. Then, assuming the existence of one-way functions secure against non-uniform probabilistic polynomial time adversaries, there is a memory checker with secret state with completeness $c - 1/n^2 - \mathsf{negl}(n)$, soundness s, and local space $p + n^{1/10}$ that supports up to T logical queries and has physical database size $\mathsf{poly}(n)$, with all other parameters unchanged.*

Proof. We use a pseudorandom function (PRF) family $\mathsf{PRF}_k : \{0,1\}^{\lceil \log_2(m) \rceil} \to \{0,1\}^{2\lceil \log_2(Tqn) \rceil}$ indexed by uniformly random key $k \sim \{0,1\}^\lambda$ where $\lambda = n^{1/10}$. Since $m \leq 2^{\mathsf{poly}(n)} = 2^{\mathsf{poly}(\lambda)}$ and $T, q \leq \mathsf{poly}(n) = \mathsf{poly}(\lambda)$, by standard reductions from one-way functions [HILL99, GGM86], we know that such a PRF family exists that is secure against non-uniform polynomial time adversaries in λ (and thus in n as well).

We will add this key k of the PRF to the local secret state of the memory checker. For all physical locations $j \in [m]$ written on the write-only tape used for physical queries, the memory checker will replace it with $\mathsf{PRF}_k(j)$ (using the natural binary mapping between $\{0,1\}^\ell$ and $[2^\ell]$ as needed). We emphasize that even if the original memory checker had public state, this new memory checker needs secret state to store the key k.

Soundness of this memory checker is immediately preserved, as the adversary can easily simulate the PRF mapping. Completeness is preserved as long as none of the physical locations used overlap. Since the memory checker supports T logical queries, the memory checker makes at most Tq physical queries, so the number of distinct physical locations that it touches is at most Tq. If the PRF were replaced by a random function, we know by a union bound that the probability that there exists a PRF collision among these at most Tq distinct locations is at most

$$\binom{Tq}{2} \cdot \frac{1}{(Tqn)^2} < \frac{1}{n^2}.$$

Therefore, by a hybrid argument using PRF security, since the adversary is efficient and only gets oracle access to the PRF, completeness deteriorates at most by an additive $1/n^2 + \mathsf{negl}(\lambda) = 1/n^2 + \mathsf{negl}(n)$ term. Furthermore, the physical database size is now at most $4(Tqn)^2 = \mathsf{poly}(n)$. □

C Write-Optimized Memory Checker Construction

In Corollary 2, we showed that for reasonable parameter settings of memory checkers, if the read query complexity is small, then the write query complexity must be large. In Corollary 3, we show that for *deterministic and non-adaptive* memory checkers, then the reverse is true, namely that when the write query complexity is small, then the read query complexity must be large.

In this section, we give (weak) evidence that for *adaptive* memory checkers, the story may be more complicated. Explicitly, we give a generic transformation that turns any memory checker into one that has very efficient writes, at the expense of deferring the queries of those logical writes to subsequent logical reads.

Theorem 10. *Assume there exist one-way functions. Suppose there is a binary memory checker for a logical memory of size n that has write query complexity q_w, read query complexity q_r, local space p, physical word size w, completeness c, soundness s, and public database size m. Then, there is a binary memory checker with secret state for a logical memory of size n supporting a fixed* $\mathsf{poly}(n)$ *number of queries with local space $p + O(n^{1/10})$, physical word size $w + \omega(\log n)$, completeness c, soundness $s + \mathsf{negl}(n)$, and public database size $m + \mathsf{poly}(n)$ with the following query complexities:*

- *The write query complexity is* 1*, and*
- *The read query complexity is $O(\ell q_w + q_r)$, where ℓ denotes the number of consecutive logical writes immediately preceding the given logical read.*

Proof. The idea for the construction is direct. The new memory checker M' will run the old memory checker M, except that for each logical write, M' will simply authenticatedly store the logical write instruction (using one query) on a public stash. When the next logical read occurs, M' will perform all of the writes that were written on the stash, reset the stash, and then perform the logical read.

More specifically, for each logical write $(i, \mathsf{data}) \in [n] \times \{0,1\}$, if this is the jth logical write since the last logical read, then M' will write $(i, \mathsf{data}, \mathsf{PRF}_k(j, i, \mathsf{data}))$ to a new public stash at index $j \in [\mathsf{poly}(n)]$, where PRF is a pseudorandom function with key size $n^{1/10}$ and output length any $\omega(\log n)$ (which we know exists assuming the existence of one-way functions).[11]

For each logical read $i' \in [n]$, M' will first iterate over all non-empty locations $j \in [\mathsf{poly}(n)]$ of the stash with contents $(i, \mathsf{data}, \sigma)$, verify that $\sigma = \mathsf{PRF}_k(j, i, \mathsf{data})$, and perform logical write (i, data) according to M. After clearing the stash and sampling a new fresh PRF key, M' then performs logical read i' according to M. The secret state of the memory checker consists of the PRF key k and the current counter j of consecutive logical writes since the last logical read.

The efficiency of M' following immediately from our construction, and completeness similarly follows. It remains to show soundness of M'.

Soundness. We provide a (detailed) sketch. Suppose there is a PPT adversary that breaks soundness of M' with probability δ. By applying the security of the PRF, we know we can replace

[11] More generally, any message authentication code (MAC) would suffice, but we use a PRF for concreteness.

it with a truly random function, at the cost of an additive $\mathsf{negl}(n)$ loss. Next, since we sample a new key for each new logical read and since we always increment j when using the PRF, we know that the probability that the adversary can pass the $\sigma = \mathsf{PRF}_k(j, i, \mathsf{data})$ check with an *incorrect* i or data is $1/2^{\omega(\log n)} = \mathsf{negl}(n)$. Therefore, with $1 - \mathsf{negl}(n)$ probability, M' is exactly running M on the non-stash portion of public memory. This soundness attack on M' is therefore an attack on M as well with probability $1 - \mathsf{negl}(n)$. Therefore, M' is $s + \mathsf{negl}(n)$ sound. □

We now apply Theorem 10 to show that there is a regime in which writes can have query complexity $O(1)$ and reads can have query complexity $O(\log n / \log \log n)$.

Corollary 7. *Assume there exist one-way functions. Consider sequences of logical queries that have no more than $O(1)$ consecutive writes. For such sequences, there is a binary memory checker with secret state for a logical memory of size n that has write query complexity $q_w = 1$, read query complexity $q_r = O(\log n / \log \log n)$, local space $O(n^{1/10})$, physical word size $\mathsf{polylog}(n)$, completeness 1, and soundness $\mathsf{negl}(n)$.*

Proof. We can directly apply the transformation of Theorem 10 to a $O(\log n / \log \log n)$-query complexity memory checker with secret state, e.g., Theorem 2 of [PT11] with PRF key size $O(n^{1/10})$ and output length $\omega(\log n)$. □