Memory Checking for Parallel RAMs

Surya Mathialagan[*]
MIT
smathi@mit.edu

November 2, 2023

Abstract

When outsourcing a database to an untrusted remote server, one might want to verify the integrity of contents while accessing it. To solve this, Blum et al. [FOCS '91] propose the notion of *memory checking*. Memory checking allows a user to run a RAM program on a remote server, with the ability to verify integrity of the storage with small local storage.

In this work, we define and initiate the formal study of memory checking for *Parallel RAMs* (PRAMs). The parallel RAM model is very expressive and captures many modern architectures such as multi-core architectures and cloud clusters. When multiple clients run a PRAM algorithm on a shared remote server, it is possible that there are concurrency issues that cause inconsistencies. Therefore, integrity verification is even more desirable property in this setting.

Assuming only the existence of one-way functions, we construct an online memory checker (one that reports faults as soon as they occur) for PRAMs with $O(\log N)$ simulation overhead in both work and depth. In addition, we construct an offline memory checker (one that reports faults only after a long sequence of operations) with amortized $O(1)$ simulation overhead in both work and depth. Our constructions match the best known simulation overhead of the memory checkers in the standard single-user RAM setting. As an application of our parallel memory checking constructions, we additionally construct the first *maliciously secure oblivious parallel RAM* (OPRAM) with polylogarithmic overhead.

[*]The author was supported in part by DARPA under Agreement No. HR00112020023, an NSF grant CNS-2154149, a grant from the MIT-IBM Watson AI, a grant from Analog Devices, a Microsoft Trustworthy AI grant, and a Thornton Family Faculty Research Innovation Fellowship from MIT. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

Contents

1 Introduction

Consider a large database outsourced to an untrusted remote storage server. A fundamental cryptographic property one might hope to achieve in this setting is *integrity verification*, i.e., the ability to verify that the server has not tampered with the contents of the storage. For example, if a hospital stores its patients' medical records on a database, the reliability of the records is crucial. Moreover, the use of cloud servers to store personal information (e.g. email, digital photographs, etc.) is widespread. For all of these applications, it is important to guarantee the integrity of the contents of the storage.

In the setting where a user outsources a *static* database, they can simply authenticate the database to ensure integrity. However, when the user outsources a database which also has to *dynamically* support updates, integrity verification becomes more complicated. This is in fact the problem of *memory checking*, which was first introduced by Blum, Evans, Gemmel, Kannan and Naor [BEG+91].

In the memory checking setting, a user $\mathcal{U}$ would like to run a RAM program on a remote storage $\mathcal{S}$. A memory checker is a layer between the user $\mathcal{U}$ and remote storage $\mathcal{S}$, as shown in Figure 1a. The user $\mathcal{U}$ sends read and write requests to $\mathcal{M}$, and $\mathcal{M}$ then sends its own read and write requests to the unreliable storage $\mathcal{S}$. The checker $\mathcal{M}$ then uses the responses from the server and its own small private local storage to determine if $\mathcal{S}$ responded correctly and send the correct response to $\mathcal{U}$. If $\mathcal{S}$ sends an incorrect response, the checker $\mathcal{M}$ reports that $\mathcal{S}$ was faulty and aborts.

There are two main efficiency metrics for memory checking: the *work blowup* (the ratio of the number of physical accesses by $\mathcal{M}$ per underlying logical query made by $\mathcal{U}$), and the *space complexity* of the local private storage of $\mathcal{M}$. Using an authentication tree [Mer90, BEG+91], it is possible to achieve $O(\log N)$ work blowup with $O(1)$ word space complexity.

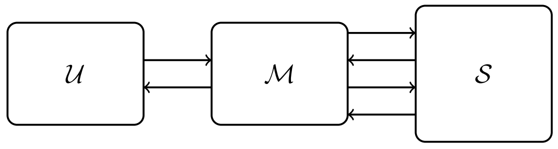
The memory checking model of Blum et al. has been well studied [BEG+91, Ajt02, NR09, DNRV09, PT11] and has found many real-world and theoretical applications. For example, many secure enclaves such as AEGIS and Intel SGX [CD16] support the integrity verification of external memory. On the theoretical side, memory checking has been used to obtain proofs of retrievability [SW13] and maliciously secure oblivious RAM (ORAM) constructions [RFY+13, MV23].

Integrity verification with multiple users. One can also ask if integrity verification can be done in a setting where there are multiple users executing a *parallel RAM* (PRAM) algorithm on a shared remote storage. The PRAM model is a generalization of the RAM model that allows for parallel batches of operations to be made to the server. The PRAM model captures many emerging technologies. For example, it can model multiple users sharing a common cloud server to perform some shared computation, or multiple processors running within a single multicore system. One can also imagine multiple entities (e.g. hospitals) sharing a single shared database that they dynamically and independently update. Due to its generality, many recent works have studied cryptography in the PRAM setting, such as Oblivious PRAM (OPRAM), Garbled PRAM, and more [BCP16, CCC+16, CLT16, LO17].

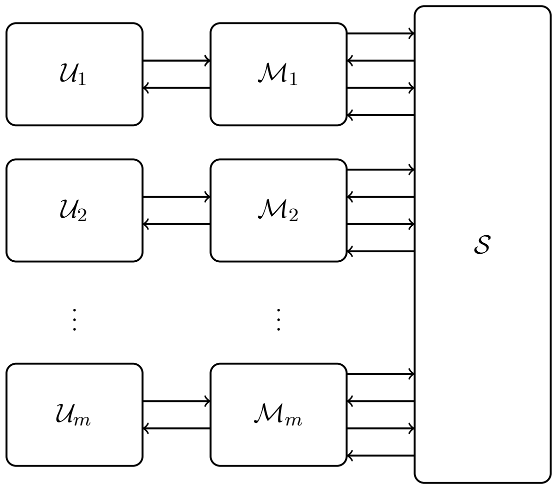
In the parallel setting, integrity verification can also be useful to ensure that the various entities have a *consistent* and most up-to-date view of remote storage. Therefore, it seems natural to extend memory checking to the parallel setting.

1.1 Our Contributions

In this work, we initiate the formal study of memory checking for PRAM programs. We first define memory checking notions for PRAMs by generalizing the definitions of Blum et al. Throughout this section, N is the size of shared remote storage with word size w, and $1 \leq m \leq N$ is the number of users.



(a) Memory checking model for RAMs as defined by Blum et al. [BEG+91]. Here, user $\mathcal{U}$ is accessing a remote storage $\mathcal{S}$. Memory checker $\mathcal{M}$ is a layer between $\mathcal{C}$ and $\mathcal{S}$ that ensures the correctness of the responses from $\mathcal{S}$.



(b) Memory checking model for PRAMs. Here, $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_m$ are CPUs that simultaneously access a server $\mathcal{S}$. To ensure the correctness of the server's responses, we have memory checkers $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_m$ as an interface for the clients $\mathcal{U}_1, \ldots, \mathcal{U}_m$, and the memory checker communicates with the server $\mathcal{S}$ to ensure the correctness of the server's responses.

Figure 1: Memory checking models for RAMs and PRAMs.

In this model, we assume that each user $\mathcal{U}_i$ interfaces with a checker $\mathcal{M}_i$ to interact with the server (see Fig. 1b). For every batch of *logical* queries from $\{\mathcal{U}_i\}_i$, the checkers $\{\mathcal{M}_i\}_i$ produce batches of *physical* requests to the $\mathcal{S}$. While the checkers $\{\mathcal{M}_i\}_i$ can use shared private randomness to generate a secret state (e.g. authentication keys) before the start of the memory checking protocol, they are not allowed to communicate directly to each other after the start of the protocol. These users can still communicate with each other through the server in an authenticated manner. This is the most general setting because it does not require any reliable communication channels between the checkers. Note that however, our model does not prevent the users $\mathcal{U}_i$ from communicating with each other, but it is general enough to accomodate users that do not communicate with each other. We formalize this model in Section 4.

We focus on three main PRAM models: exclusive-read exclusive-write (EREW), concurrent-read exclusive-write (CREW) and concurrent-read concurrent-write (CRCW). In the first model, we assume that at most one user accesses any index of the remote storage at any parallel time step. In the CREW model, we allow concurrent read accesses to any location, but only one user can access any given location for a write. Finally, in the CRCW model we allow concurrent reand and write accesses to memory locations, where we resolve write-conflicts according to some pre-determined rule (e.g. an arbitrary user wins any write. See Section 3 for more examples). Our results apply to most natural conflict-resolution rules.

Efficiency metrics. Like Blum et al., we are interested in minimizing *work blowup* (i.e. the ratio of the number of physical queries for every batch of logical queries) and *space complexity* of each $\mathcal{M}_i$. Moreover, an additional complexity measure we hope to minimize in the case of PRAMs is *depth* or *parallel complexity blowup*. In other words, for each parallel batch of instructions from the users $\{\mathcal{U}_i\}_i$, we hope to minimize the number of parallel batches of instructions from $\{M_i\}_i$. For all constructions in this work, the blowup in server space storage is $O(1)$.

The problem with concurrency. As we detail in Section 2, allowing concurrent reads and writes makes the memory checking problem more challenging. In the standard RAM setting (as in Fig. 1a), the problem of memory checking boils down to checking whether a server returns corrupted data when $\mathcal{U}$ performs a read. In the CRCW PRAM setting, we also run into problems concurrency issues with writes. If multiple users attempt to write to the same address, $\mathcal{S}$ essentially gets to choose which user wins the write. However, nothing prevents $\mathcal{S}$ from pretending that multiple different writes were accepted. For example, if $\mathcal{U}_1$ and $\mathcal{U}_2$ both write to some address addr, the server can now branch the memory into two versions - one with $\mathcal{U}_1$ winning the write, and one with $\mathcal{U}_2$ winning the write. Therefore, we need to ensure that $\mathcal{S}$ does not branch the memory and instead commits to a single consistent memory across all users.

Online memory checking for PRAMs. The notion of memory checking defined above is known as *online memory checking* since no $\mathcal{M}_i$ sends incorrect responses to its user, i.e. the correctness of responses is ensured in an *online* manner. In particular, if the server sends an incorrect response to some $\mathcal{M}_i$, there exists some $\mathcal{M}_j$ (possibly different from $\mathcal{M}_i$) which will abort before $\mathcal{M}_i$ sends any response to $\mathcal{U}_i$.

In the standard single-user RAM setting, one can implement online memory checkers with collision-resistant hash functions (CRHFs) following the Merkle-tree paradigm [Mer90] with $O(\log N)$ work blowup. Blum et al. [BEG$^+$91] show a tree-based memory checker with $O(\log N)$ overhead which can be instantiated with message authentication codes (MACs). At a high level, the construction maintains a binary tree where the leaf nodes correspond to the elements of the underlying database. Every leaf node is given a counter value keeping track of the number of times the associated database entry is updated. Every non-leaf node contains the sum of the counts of its children nodes. $\mathcal{M}$ keeps track of the value at the root node. Every node is authenticated, and any read to any node is verified. If $\mathcal{U}$ performs a write to some entry, the counts along the corresponding path are incremented. If $\mathcal{U}$ performs a read to some database entry, $\mathcal{M}$ traverses the path to the corresponding leaf node. While doing so, $\mathcal{M}$ verifies the consistency of the counts of the nodes on the path. Since every node is authenticated, one can argue that any adversarial behavior from the server can be detected because either the authentication verification fails, or the counts are inconsistent. Since the binary tree has $O(\log N)$ height, this introduces an $O(\log N)$ simulation overhead. We refer the reader to Section 2 for more details on this construction.

There is no known construction beating the $O(\log N)$ overhead. Moreover, Dwork, Naor, Rothblum and Vaikuntanathan [DNRV09] showed a $\Omega(\log N/\log\log N)$ lower bound on the blowup for memory checkers which are deterministic and non-adaptive - capturing most known memory checkers. Therefore, this is essentially the best work blowup one can hope for.

One can imagine that by serializing a given PRAM algorithm (i.e. at each time-step, exactly one user accesses the server), one can adapt a tree-based online memory checker such as the construction of Blum et al. [BEG+91] or a Merkle tree [Mer90]. However, this gives a memory checking construction with $O(\log N)$ work blowup and $O(m \log N)$ depth blowup. While the work blowup matches that of memory checking for RAMs, the depth blowup is in fact equal to the total work, and does not capitalize on the parallelization capabilities of our model. Therefore, one can ask if it is also possible to also achieve an $O(\log N)$ depth blowup. In this work, we show that this is indeed possible.

Theorem 1.1 (Informal version of Theorem 5.4)**.** *Assuming the existence of one-way functions, there exists an online memory checking protocol with $O(\log N)$ work blowup, $O(\log N)$ depth blowup and $O(1)$ local space complexity per checker.*

We remark that if the underlying algorithm is EREW or CREW, the access pattern of the resulting memory checking protocol is also EREW or CREW respectively when interacting with an honest server.

Naor and Rothblum [NR09] show that one-way functions are in fact necessary for online memory checking for RAM programs, and hence this assumption is also necessary for our result. Moreover, when we consider the special case where $m = 1$, our result reduces to memory checking for RAM programs, and our efficiency in fact matches the best known memory checkers [BEG+91, DNRV09, PT11].

The starting point of our construction is the authentication tree of Blum et al. [BEG+91]. However, there are two main technical difficulties that arise when directly implementing their construction. Firstly, since multiple elements of the database might be accessed in the same batch of queries, this could result in many read-write conflicts when updating the internal nodes of the tree. Secondly, if the underlying algorithm performs concurrent accesses, as previously mentioned, the server could potentially branch the history by showing multiple incompatible versions of the storage to different users. Therefore, we have to ensure that all the users view exactly one consistent copy of the authentication tree.

To solve the first problem, we simply introduce a simple tie-breaking rule. If two CPUs in charge of two children nodes want to update the parent node at the same time, we give the left node priority. This ensures that in an honest execution, at most one CPU attempts to update any given internal node of the authentication tree. To avoid the branching-history problem, we use a counting technique. Essentially, in addition to updating the counters of the leaf nodes of the authentication tree, each CPU also locally keeps track of whether it successfully executed a write (i.e. if its write won the conflict resolution rule). Once the counts are propagated through the the authentication tree, we can then verify that the number of successful writes recorded at the root node corresponds to the total number of CPU writes. If the server tells more than one client that they "won", we argue that we will detect a discrepancy. We discuss our techniques in further detail in Section 2.

Offline memory checking for PRAMs. Blum et al. [BEG+91] also suggest a weaker notion of memory checking known as *offline* memory checking. An offline checker gives a weaker guarantee that after a long sequence of operations to the storage, it can detect whether there was any faulty response from the storage. To contrast with online memory checking, we note that it is possible that some $\mathcal{M}_i$ sends back an incorrect response to $\mathcal{U}_i$, but by the end of the algorithm, with high probability, some $\mathcal{M}_j$ (not necessarily the same as $\mathcal{M}_i$) reports that some mistake has occurred.

Model	CPUs	Total Work	Total Parallel Depth	Assumption	Reference
RAM	1	$O(q+N)$	-	None	[BEG+91, DNRV09]
RAM	1	$O(q+N)$	-	OWF	[MV23]
EREW	m	$O(q+N)$	$O(d+N/m+\log m)$	None	Theorem 6.4
CRCW/CREW	m	$O(q+N+dm\log m)$	$(d\log m+N/m+\log m)$	None	Corollary 6.5
CRCW/CREW	m	$O(q+N)$	$O(d+N/m+\log m)$	OWF	Theorem 7.3

Table 1: Consider offline checking for a storage of size N with a m-user database. Here, we are given an underlying PRAM program with q queries and depth d over a database of size N, and the table represents the work and parallel complexity of the communication with the remote storage after applying an offline memory checker.

The main benefit of an offline checker is that it is possible to achieve an amortized work blowup of $O(1)$. In fact, Blum et al. showed that there exists a statistically secure offline memory checker with amortized $O(1)$ query complexity, i.e. even a computationally unbounded remote server $\mathcal{S}$ cannot fool the memory checker with high probability. To achieve this, they use ϵ-biased hash functions as constructed by Naor and Naor [NN90]. The work of Arasu et al. [AEK+17] alludes to the fact that this algorithm is parallelizable for EREW programs[1]. We give a formal exposition of the algorithm and prove that this is in fact the case.

Theorem 1.2 (Informal version of Theorem 6.4)**.** *There exists a statistically secure offline memory checker for EREW PRAM algorithms with amortized $O(1)$ blowup in work and parallel complexity.*

Since all CREW and CRCW programs can be emulated in the EREW model with logarithmic overhead in work and parallel complexity, this additionally gives us a statistically secure memory checker for CREW and CRCW PRAM programs as well. However, the amortized blowup of such a scheme is $O(\log m)$ in terms of work and depth.

To achieve $O(1)$ amortized complexity, we instead draw inspiration from the offline memory checking construction of Mathialagan and Vafa [MV23] which relies on authentication.

Theorem 1.3 (Informal version of Theorem 7.3)**.** *Assuming the existence of one-way functions, there exists an offline memory checker with amortized $O(1)$ work blowup and amortized $O(1)$ depth blowup.*

The main difficulty in obtaining this result once again is ensuring that the adversary does not branch the memory (i.e. by accepting different concurrent writes from the perspective of multiple users). To resolve this issue, we carefully extend our counting argument from our online memory checking construction to the offline setting as well. Additionally, authentication seems to be necessary to

[1]Arasu et al. [AEK+17] ultimately instantiate the algorithm of Blum et al. [BEG+91] with pseudorandom functions.

prevent any "spoofing" attacks from the server. We elaborate our techniques in Section 2. We state the exact work and parallel overhead of our offline checkers in Table 1.

Relaxing the parallelization requirements. Just like the RAM model, the PRAM model is rather idealized and abstracts out many practical considerations such as sychronization. In the respective sections (Remarks 5.3 and 7.2), we argue that the memory checking algorithms are flexible and can in fact be generalized to work with some notion of "rounds" without the need for synchronization.

Application to Oblivious Parallel RAM (OPRAM). Oblivious RAM is a primitive which takes a sequence of RAM queries to a server and transforms the access pattern to remove any information leakage to the server. As a general technique to ensure privacy of RAM computations, ORAM has many applications including cloud computing, multi-party protocols, secure processor design, and private contact discovery, the latter as implemented by the private messaging service Signal [SCSL11, FDD12, LO13, WHC+14, BNP+15, FRK+15, GHJR15, LWN+15, ZWR+16, DFD+21, Con22].

Boyle, Chung and Pass [BCP16] extended this notion to the parallel RAM setting, and defined the notion of an *Oblivious Parallel RAM* (OPRAM). OPRAM is a compiler that allows multiple users to interact with a remote server concurrently in a privacy-preserving way. After a series of works [BCP16, CCS17, CGLS17, CS17], the work of Asharov et al. [AKL+22] constructed an OPRAM with $O(\log N)$ blowup.

Theorem 1.4 (Informal, [AKL+22])**.** *Assuming the existence of one-way functions, there exists an arbitrary CRCW OPRAM scheme with $O(\log N)$ blowup in both work and depth.*

Asharov et al. [AKL+22] additionally show that their construction is optimal when the number of CPUs $m = O(N^{0.99})$.

However, this OPRAM construction is only known to be secure in the *honest-but-curious* setting, where the adversary answers all read and write queries *honestly*. In reality, the adversary can do a lot more. If the adversary tampers with the database contents and returns corrupted responses, the OPRAM scheme may no longer be secure. We say that an OPRAM is maliciously secure if it is secure even against tampering adversaries.

Several works [GO96, RFY+13, MV23] have noted that composing memory checkers with ORAM constructions is sufficient to obtain malicious security. By a similar argument, we can combine our PRAM memory checker with the optimal OPRAM of [AKL+22] to obtain the following result.

Theorem 1.5 (Informal version of Theorem 8.11)**.** *Assuming the existence of one-way functions, there exists a maliciously secure arbitrary CRCW OPRAM scheme with $O(\log^2 N)$ blowup in both work and depth.*

To the best of our knowledge, this is the first maliciously secure OPRAM construction with polylogarithmic overhead.

In the case of ORAMs, Mathialagan and Vafa [MV23] were able to intricately interleave offline and online memory checking for RAMs with the optimal ORAM construction of Asharov et al. [AKLS21] to avoid the additional log factor from memory checking, and obtained a maliciously secure ORAM with optimal logarithmic overhead. We believe that our offline and online PRAM memory checking constructions can also be similarly used to obtain a more efficient maliciously secure OPRAM. We leave this for future work.

1.2 Related Work

We will compare our model and results to some related work.

Byzantine agreement and distributed consensus. Our model differs from the traditional distributed algorithms setting for Byzantine agreement [PSL80] crucially because our model has no reliable communication channels between the users. The only way the users can communicate with each other in our setting is through an unreliable remote server. We also assume that the users are trusted. On the other hand, the focus in many works in Byzantine agreement and consensus [PSL80, DS83, FL82, FLM86] in the presence of faulty/malicious users. For example, in work of Dolev and Strong [DS83], all communication channels are thought to be *reliable* (i.e. no spoofing attacks), but authentication is still useful in ensuring malicious users cannot introduce new messages in the information exchange.

Parallel memory checking. In the work of Papamanthou and Tamassia [PT11], they consider the parallel complexity of memory checking for a RAM program. In other words, a single user makes an update to the remote storage, but the memory checker itself is able to send parallel batches of requests to the remote storage. Instead, our work focuses on allowing many users to concurrently access a shared database.

1.3 Organization

In Section 2, we discuss the main technical challenges of memory checking with concurrency, and give an overview of our memory checking algorithms. In Section 3, we define the RAM and PRAM models, and introduce cryptographic primitives that we use in our construction. In Section 4, we formally define the memory checking model for parallel RAMs. In Section 5, we give our online memory checking construction. In Section 6, we construct a statistically-secure offline memory checker for EREW algorithms with amortized $O(1)$ complexity. In Section 7, we construct a computationally-secure offlime memory checker for CRCW with amortized $O(1)$ complexity. In Section 8, we show how we can apply our memory checking techniques to obtain a maliciously secure oblivious parallel RAM construction with logarithmic overhead.

2 Technical Overview

In this section, we give an overview of our constructions in the EREW setting. We then highlight the core difficulties that arise in the CRCW setting due to the concurrency, and describe how we deal with these issues.

2.1 Overview of our constructions

First, we give an overview of our algorithms. For simplicity, we first consider the case where the underlying PRAM program satisfies the EREW model.

Authentication trees. For our online memory checking construction, we follow the authentication tree paradigm for RAM models [Mer90, BEG$^+$91]. We first recall the memory checking construction of [BEG$^+$91] for RAMs. At a high level, an authentication tree stores the database at the leaves of a binary tree. At the leaves, the version number (i.e. the number of updates made) of every element is stored along with contents of the memory location. The parents of the leaf nodes then contain the sum of the version numbers of its two children. Every subsequent internal node contains the sum of the counts on both of its children. Every node is authenticated. The memory checker then keeps track of the count stored at the root node at any point in time. The main invariant maintained is that if the server functions honestly, then the count stored at any internal node is the sum of the counts of its two children.

When a user wants to read a memory location, the checker verifies the counts of all the elements from the root to the corresponding leaf node, and ensures that the count is in fact the sum of the counts of its children. When a user writes to a memory location, it increments the counts of all nodes on the path from the leaf to the root. At a high level, this is secure because by the security of MACs, the server $\mathcal{S}$ can only present "stale" values with lower counts. Since we know the count of the root node reliably, one can always detect a replay attack.

Authentication trees for PRAMs. When extending this construction to EREW or CREW PRAMs, we run into the issue that if many leaf values are updated in parallel, there will be many conflicts at the internal nodes of the tree. If we serialize the updates (i.e. one update is made at a time), the depth complexity of the algorithm blows up by $O(m \log N)$.

In order to update the tree in parallel, we introduce a tie-breaking rule to ensure at most one checker updates any internal node in the tree. For EREW/CREW algorithms, clearly at most one checker updates any leaf node. After updating the leaf nodes, we now have to propagate the updated counts to the rest of the tree. To ensure that the nodes at the next level have a unique CPU assigned, we always give priority to the CPU that updated the left child. In other words, the CPU associated to the right child first checks if the left child was updated (e.g. by checking a time-stamp). If so, the CPU in charge of the left node is now in charge of the parent node. Otherwise, the CPU in charge of the right node is now in charge of the parent. We use a similar rule to assign a checker to any parent. It is clear that this can be done in an EREW manner with $O(1)$ blowup in time-complexity.

At the end of each iteration, the algorithm then tallies the number of checkers that made updates to the database at a given time-step (can be done in $O(\log m)$ depth), and verifies that the root node count has in fact increased by that amount. This ensures that all internal nodes were in fact increased consistently. For a full exposition of this algorithm, see Section 5.

Offline memory checking. We now describe our offline memory checking construction from one-way functions, once again in the context of EREW algorithms. We draw inspiration from the counting-based argument of Mathialagan and Vafa [MV23].

At a high level, every memory location stored on the server is tagged with a version number (i.e. the number of times that element was udpated). Whenever a checker reads or writes to a memory location, it writes back to the memory location with the version number incremented. The checker also locally increments a counter. Note that we authenticate every read and write to this server. In the offline setting, since reads and writes both result in updates to a memory location, the CREW model and CRCW model both have CRCW offline-checkers.

At the end of the sequence of operations, the checkers sum the version numbers of all the elements on the server, and compares this to the sum of the local counters of all the checkers. By the security of MACs, we have that the sums are equal only if the server succeeds in forgery or if the server did not corrupt any of the responses. For a full exposition and proof of correctness, see Section 7.

2.2 Main Challenges with Concurrency

We now describe the subtleties that arise in the CRCW model that do not show up in the RAM or EREW/CREW PRAM model.

Concurrent reads and writes. In the *arbitrary* CRCW PRAM model, multiple users are able to write to the same location at any point in time. For example, suppose both $\mathcal{U}_1$ and $\mathcal{U}_2$ try to write values v_1 and v_2 respectively to some address addr. Then, a malicious storage server $\mathcal{S}$ could essentially branch the storage into two states: a state where location addr contains v_1, and a state where location addr contains v_2 instead. Therefore, our memory checking protocol must account for this, and force the server to commit to one consistent memory[2]. Note that this may not be a problem for conflict resolution rules such as priority CRCW which uniquely determines the CPU that "win" the concurrent write.

Preventing spoofing attacks. On the other hand, one can also imagine that a server could block a memory location addr that some user $\mathcal{U}_i$ wishes to update, by "spoofing" some other user $\mathcal{U}_j$. Therefore, the server can repeatedly do this and block every memory location. However, this attack can be prevented by using authentication. This fundamentally seems to be the reason we are unable to obtain a statistically secure CRCW offline memory checker with $O(1)$ amortized work blowup.

Note that both of the above attacks do not appear in the EREW PRAM algorithms, since every memory checker knows that there will be no conflict during a read or a write.

2.3 Our Techniques for Concurrency

Although dealing with memory branching seems like a daunting task, we show how one can use authentication along with a simple counting argument to prevent branching in both our offline and online memory checking constructions. We give a high-level overview of our counting technique.

[2]In other words, we want a single "sacred timeline" that every user sees, in the language of the Marvel television series *Loki*.

In both of our online and offline checking constructions, every address is tagged with a version number count, initialized to 0 at the start of the algorithm. This version number keeps track of the number of times the location was accessed. We instantiate every write in a few phases. At parallel time-step t, we do the following:

1. *Test phase:* First, every user $\mathcal{U}_i$ reads from their desired location addr to retrieve the version number count of the location. Then, every $\mathcal{M}_i$ attempts to write to its desired address, with a test flag set. The $\mathcal{M}_i$ also tags their data with the time-step t, user ID i and increases the version number of count.
2. *Winner phase:* Now, every checker reads the same location again to determine if they "won" the concurrent write.
   - If $\mathcal{U}_i$ in fact "won" the write (i.e. the storage $\mathcal{S}$ reports back with their write), $\mathcal{M}_i$ writes back the contents with the test flag set to false.
   - If $\mathcal{U}_i$ did not "win" the write, $\mathcal{M}_i$ ensures that the winning user $\mathcal{U}_j$ in fact set the test flag to false, set the time-step to t, and has a consistent count value.

By authenticating all writes with a shared secret MAC key, one can be sure that the server is not spoofing fake writes. Additionally, the verification of the "test" flag ensures that there are no cycle of winners. In other words, we prevent the scenario where user U_{i_1} receives the signal that U_{i_2} won, U_{i_2} receives the signal that U_{i_1} won. While this prevents a spoofing attack, this does not yet prevent branching in memory.

Throughout the algorithm, every $\mathcal{M}_i$ keeps track of the number of concurrent writes it has won. Let C be the sum of the highest version numbers of all memory locations, and let M be the sum of the number of concurrent writes won by all $\mathcal{M}_1, \mathcal{M}_2, \ldots, \mathcal{M}_m$. Assuming unforgeability of MACs, our key observation is that $C = M$ if and only if the server responded with respect to a "consistent version" of the storage to all users. In fact, if the server lies at any point, it must be the case that at the end of the algorithm, $C < M$. We formalize this argument in the proofs of Theorem 5.4 and Theorem 7.3.

3 Preliminaries

Throughout this work, we let λ be the security parameter. In all of these constructions, we assume the adversary or the server $\mathcal{S}$ runs in time $\mathsf{poly}(\lambda)$. We say that a function $\mathsf{negl} : \mathbb{N} \to \mathbb{R}^+$ is negligible if for every constant c, there exists N_c such that $\mathsf{negl}(\lambda) < \lambda^{-c}$ for all $\lambda > N_c$. For an integer $n \in \mathbb{N}$, we denote by $[n]$ the set $\{1, 2, \ldots, n\}$. We use the notation (x, y) or $(x||y)$ to indicate string concatenation of x and y.

3.1 Parallel RAM Machines

RAM machines. A RAM is an interactive turing machine with memory mem containing N logical addresses where each memory cell indexed by $\mathsf{addr} \in [N]$ contains a word of size w. The RAM supports read and write operations. Read operations are of the form $(\mathsf{read}, \mathsf{addr}, \perp)$ where $\mathsf{addr} \in$

$[N]$, the RAM returns the contents of $\mathsf{mem}[\mathsf{addr}]$. Write operations are of the form $(\mathsf{write}, \mathsf{addr}, v)$, in which case the RAM updates the contents of $\mathsf{mem}[\mathsf{addr}]$ to be v. As standard in previous works, we assume that word-level addition, Boolean operations and evaluating PRFs can be done in unit cost.

In this work, we generally set $w = \omega(\log \lambda)$. While this is not standard for the RAM model, many memory checking constructions implicitly operate in this setting since most construction use MACs or CRHFs, which need to be of size $\omega(\log \lambda)$ to be secure against $\mathsf{poly}(\lambda)$ adversaries. For a detailed discussion, see Section 2.4 of Mathialagan and Vafa [MV23].

Parallel RAM machines. A parallel RAM (PRAM) is a generalization of a RAM but with multiple CPUs. In fact, a RAM is simply a PRAM with exactly 1 CPU. A PRAM comprises m CPUs and a shared memory mem containing N logical addresses where each memory cell indexed by $\mathsf{addr} \in [N]$ contains a word of size w. Just like RAMs, we assume the word-level operations such as word-level addition, Boolean operations and evaluating PRFs can be done in $O(1)$ time.

At time step t of the execution, each CPU might compute some request $\vec{I}_i^{(t)} = (\mathsf{op}, \mathsf{addr}, \mathsf{data})$. Then, the RAM receives a sequence of requests $\vec{I}^{(t)} := (I_i^{(t)} : i \subseteq [m])$ (i.e. a set of requests from a subset of the CPUs). If $\mathsf{op}_i = \mathsf{read}$, then CPU_i receives the contents of $\mathsf{mem}[\mathsf{addr}_i]$ at the start of time-step t. If $\mathsf{op}_i = \mathsf{write}$, then the contents of $\mathsf{mem}[\mathsf{addr}_i]$ are updated to data_i.

Write conflict resolution. In the PRAM model, it is possible that multiple CPUs attempt to access a given address at the same time. If a PRAM algorithm guarantees that any address is accessed by at most one CPU at any given time-step, we say that the algorithm is exclusive-read exclusive-write (EREW). If CPUs can concurrently read any address but at most one CPU writes to an address at any given time-step, we say the algorithm is concurrent-read exclusive-write (CREW).

On the other hand, if there are multiple concurrent accesses to the same address for both reads and writes, we call this the concurrent-read concurrent-write (CRCW) model. Since many CPUs can perform a write to the same address, we need a conflict resolution rule so that the PRAM update operations are well-defined. Here are a few commonly used rules:

- Arbitrary CRCW: An arbitrarily chosen CPU wins a write.
- Priority CRCW: Processors are ordered by some fixed priority, and the processor with the highest priority wins any write.
- Maximum/Minimum CRCW: The write with the maximum or minimum value is accepted.

It is well known that a CREW or CRCW algorithm can be transformed into an EREW algorithm with a logarithmic slow-down.

Lemma 3.1. *Consider a (possibly randomized) CREW/CRCW algorithm with work q and depth d on a m-processor PRAM. Such an algorithm can be converted into an EREW algorithm with work $O(q + dm \log m)$ and depth $O(d \log m)$ on an m-processor PRAM.*

Throughout our paper, we often use the following fact about the parallel runtime of adding n numbers.

Lemma 3.2. *There is an m-CPU EREW algorithm that sums n numbers with $O(n)$ work and $O(n/m + \log m)$ depth.*

This is done as follows. First, each of the m processors sums n/m of the numbers. This step takes $O(n)$ work and $O(n/m)$ depth. Then, the m CPUs publish their current m values, and these are then summed up in a binary tree fashion. This step takes $O(m)$ work and $O(\log m)$ depth.

3.2 Authentication

In this section, we define the cryptographic primitives we need, such as pseudorandom function families (PRFs) and message authentication codes (MACs). Recall that we consider the setting where the word-size $w = \omega(\log \lambda)$.

Definition 3.3 (Pseudorandom functions (PRFs)). *Let* PRF *be an efficiently computable function family indexed by keys* $\mathsf{sk} \in \{0,1\}^{\ell(\lambda)}$*, where each* $\mathsf{PRF}_{\mathsf{sk}}$ *takes as input a word* $x \in \{0,1\}^w$ *and outputs a value* $y \in \{0,1\}^w$*. A function family* PRF *is secure if for every (non-uniform) PPT* $\mathcal{A}$*, it holds that*

$$\left| \Pr_{\mathsf{sk} \leftarrow \{0,1\}^{\ell(\lambda)}} \left[\mathcal{A}^{\mathsf{PRF}_{\mathsf{sk}}(\cdot)}(1^\lambda) = 1 \right] - \Pr_{f \leftarrow F_w} \left[\mathcal{A}^{f(\cdot)}(1^\lambda) = 1 \right] \right| \leq \mathsf{negl}(\lambda)$$

for all large enough $\lambda \in \mathbb{N}$*, where* F_w *is the set of all functions that map* $\{0,1\}^w$ *into* $\{0,1\}^w$*.*

Often, we only need a PRF with smaller domain or codomain than $\{0,1\}^w$. We abuse notation and use the same PRF, where we implicitly either pad the input with 0s or ignore a suffix of the PRF output.

Definition 3.4 (MACs). *A* message authentication code (MAC) *scheme is a triple of PPT algorithms* $(\mathsf{MACGen}, \mathsf{MAC}, \mathsf{MACVer})$ *with the following syntax:*

- $\mathsf{MACGen}(1^\lambda)$*: Outputs a key* $k \in \{0,1\}^{\ell(\lambda)}$*.*
- $\mathsf{MAC}_k(m)$*: Outputs an authentication* $\mathsf{tag} \in \{0,1\}^w$ *on the key* k *corresponding to the input message* $m \in \{0,1\}^w$*.*
- $\mathsf{MACVer}_k(m, \mathsf{tag})$*: Outputs 1 or* $\perp$*.*

Moreover, the following properties should hold:

- **Correctness:** *For all* $m \in \{0,1\}^w$*, it holds that* $\Pr[\mathsf{MACVer}_k(m, \mathsf{tag}) = 1] = 1$*, where the probability is taken over* $k \leftarrow \mathsf{MACGen}(1^\lambda)$ *and* $\mathsf{tag} \leftarrow \mathsf{MAC}_k(m)$*.*
- **Unforgeability:** *For all PPT adversaries* $\mathcal{A}$ *with oracle access to* $\mathsf{MAC}_k(\cdot)$ *and* $\mathsf{MACVer}_k(\cdot,\cdot)$*, let* $S_{\mathcal{A}}$ *be the random variable corresponding to the set of* m_i *queried by* $\mathcal{A}$ *to the* $\mathsf{MAC}_k(\cdot)$ *oracle. Then, for* $k \leftarrow \mathsf{MACGen}(1^\lambda)$ *and* $\mathcal{A}^{\mathsf{MAC}_k(\cdot), \mathsf{MACVer}_k(\cdot,\cdot)}(1^\lambda) \to (m^*, \mathsf{tag}^*)$ *where* $|m^*| = w$*, we require that*

$$\Pr\left[\mathsf{MACVer}_k(m^*, \mathsf{tag}^*) = 1 \wedge m^* \notin S_{\mathcal{A}}\right] = \mathsf{negl}(\lambda).$$

We use the standard construction of MACs from PRFs (e.g., from [Gol09]), as follows. $\mathsf{MACGen}(1^\lambda)$ outputs uniformly random $k \leftarrow \{0,1\}^{\ell(\lambda)}$, $\mathsf{MAC}_k(m)$ outputs $\mathsf{tag} = \mathsf{PRF}_k(m)$, and lastly, $\mathsf{MACVer}_k(m, \mathsf{tag}) = 1$ if and only if $\mathsf{PRF}_k(m) = \mathsf{tag}$. Correctness immediately follows, and unforgeability follows from PRF security because both the MAC_k and MACVer_k oracles can be instantiated with a PRF oracle, and the probability of correctly guessing a new output from a random function is $2^{-w} = 2^{-\omega(\log \lambda)} = \mathsf{negl}(\lambda)$.

4 Memory Checking Model

In this section, we first recall the notion of memory checking for RAMs as introduced by Blum et al. [BEG+91]. We then define our notion of memory checking for PRAMs.

4.1 Memory Checking for RAMs

We recall the notion of memory checking from Blum et al. [BEG+91]. A memory checker M can be defined as a probablistic RAM program that interacts with a user $\mathcal{U}$ and server $\mathcal{S}$, where $\mathcal{U}$ is performing a RAM computation with memory held by $\mathcal{S}$. Specifically, without a memory checker, $\mathcal{U}$ sends $(\mathsf{op}, \mathsf{addr}, \mathsf{data}) \in \{\mathsf{read}, \mathsf{write}\} \times [N] \times (\{0,1\}^w \cup \{\perp\})$ to $\mathcal{S}$, who may or may not correctly follow the RAM command, i.e., may send the wrong word back to $\mathcal{U}$ when $\mathsf{op} = \mathsf{read}$. $\mathcal{M}$ now serves as an intermediary between $\mathcal{U}$ and $\mathcal{S}$ (see Fig. 1a) that takes in each query from $\mathcal{U}$ and generates and sends (possibly multiple and adaptive) queries to $\mathcal{S}$. Whenever $\mathsf{op} = \mathsf{read}$, $\mathcal{U}$ once again generates and sends (possibly multiple and adaptive) queries to $\mathcal{S}$, and $\mathcal{M}$ is then required to either respond to $\mathcal{U}$ with some word or abort by sending $\perp$ to indicate a malicious $\mathcal{S}$. Once the memory checker aborts, the protocol is done. This continues in rounds until $\mathcal{U}$ is done sending queries, of which there are at most $\mathsf{poly}(\lambda)$.

Definition 4.1 (Online Memory Checker)**.** *We say that $\mathcal{M}$ is an* online memory checker *if for any $\mathcal{U}$ the following two properties hold:*

1. **Completeness**: *If $\mathcal{S}$ is honest, then $\mathcal{M}$ never aborts and the responses that $\mathcal{M}$ sends to $\mathcal{U}$ are all correct with probability $1 - \mathsf{negl}(\lambda)$.*
2. **Soundness**: *For all p.p.t. $\mathcal{S}$, the probability that $\mathcal{M}$ ever sends some incorrect response to $\mathcal{U}$ is $\mathsf{negl}(\lambda)$. That is, for each request from $\mathcal{U}$, if $\mathcal{S}$ sends an incorrect response to $\mathcal{M}$, $\mathcal{M}$ can either independently recover the correct answer and send it to $\mathcal{U}$, or it can abort by sending $\perp$ to $\mathcal{U}$.*

We call such a memory checker "online" because the memory checker must be able to catch incorrect responses from $\mathcal{M}$ as soon they are sent. On the other hand, one can define the notion of an "offline" memory checker:

Definition 4.2 (Offline Memory Checker)**.** *We say that $\mathcal{M}$ is an* offline memory checker *for $\mathcal{U}$ if the following two properties hold:*

1. **Completeness**: *If $\mathcal{S}$ is honest, then $\mathcal{M}$ never aborts, and the responses that $\mathcal{M}$ sends to $\mathcal{U}$ are all correct with probability $1 - \mathsf{negl}(\lambda)$.*

2. **Soundness**: *For all p.p.t. $\mathcal{S}$, if $\mathcal{M}$ ever sends an incorrect response to $\mathcal{U}$, it must abort by the end of the last request made by $\mathcal{U}$ (the user indicates this by sending $\perp$ to $\mathcal{M}$, for example) with probably at least $1 - \mathsf{negl}(\lambda)$.*

In other words, $\mathcal{M}$ may send many incorrect responses to $\mathcal{U}$, but if it does, by the end of the computation, $\mathcal{M}$ must detect that there was some error at some point. We emphasize that $\mathcal{M}$ does not need to know where or when an error occurred.

In both the offline and online memory checking setting, we want the correctness and completeness requirements to hold for any request sequence and for any behavior of the unreliable memory. In particular, we consider security against a malicious adversary $\mathcal{A}$ that controls all messages sent to $\mathcal{M}$, i.e, controls both $\mathcal{U}$ and the server responses to $\mathcal{M}$.

4.2 Memory Checking for Parallel RAMs

We now define the memory checking model for PRAMs. As pictured in Fig. 1b, given a PRAM with m CPUs $\{\mathcal{U}_i\}_{i\in[m]}$, we have corresponding family of memory checkers $\{\mathcal{M}_i\}_{i\in[m]}$. Before the start of the protocol, there is a *set-up phase* where the memory checkers run a probabilistic key generation algorithm $(s_1, \dots, s_m) \leftarrow \mathsf{Gen}(1^\lambda, 1^m)$, and obtain secret states based on shared randomness. Now, each $\mathcal{M}_i$ only locally stores s_i.

Each $\mathcal{M}_i$ acts as an intermediary between $\mathcal{U}_i$ and the $\mathcal{S}$. If $\{\mathcal{U}_i\}_i$ at parallel time-step t directly sends $\vec{I}^{(t)} = (I_i^{(t)} : i \subseteq [m])$ to the server, the server may not carry out the commands correctly or *consistently.* Instead, each $\mathcal{U}_i$ now sends $I_i^{(t)} = (\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i)$ to $\mathcal{M}_i$. Now, the family $\{\mathcal{M}_i\}_i$, in parallel, make multiple (possibly adaptive) queries in parallel $\vec{I}^{(t,1)}, \vec{I}^{(t,2)}, \dots, \vec{I}^{(t,\ell_t)}$ for some $\ell_t \in \mathbb{N}$ to $\mathcal{S}$. Then, $\mathcal{M}_i$ needs to respond to $\mathcal{U}_i$ either with some word, or $\perp$ if it detects any malicious behavior from the adversary. This continues in rounds until $\{\mathcal{U}_i\}_i$ is done sending queries, of which there are at most $\mathsf{poly}(\lambda)$ batches of requests.

Definition 4.3 (Online memory checker for PRAMs)**.** *We say that the family $\mathcal{M} = \{\mathcal{M}_i\}_i$ is an parallel online memory checker family if for all CPUs $\{\mathcal{U}_i\}_i$ where each $\mathcal{M}_i$ is an intermediary between $\mathcal{U}_i$ and $\mathcal{S}$, if the following two properties hold:*

- ***Correctness:*** *If $\mathcal{S}$ is honest, then no $\mathcal{M}_i$ aborts and the responses from $\mathcal{M}_i$ to $\mathcal{U}_i$ are all correct with probability $1 - \mathsf{negl}(\lambda)$.*
- ***Soundness:*** *For all p.p.t. $\mathcal{S}$ that $\mathcal{M}_i$ sends an incorrect response to $\mathcal{U}_i$ is $\mathsf{negl}(\lambda)$. In particular, if $\mathcal{S}$ sends an incorrect response to $\mathcal{M}_i$, either $\mathcal{M}_i$ recovers the correct answer and sends it to $\mathcal{U}_i$, or some $\mathcal{M}_j$ (not necessarily the same as $\mathcal{M}_i$) aborts with $1 - \mathsf{negl}(\lambda)$ probability.*

Similarly, we define offline memory checking for PRAMs as follows.

Definition 4.4 (Offline memory checker for PRAMs)**.** *We say that the family $M = \{\mathcal{M}_i\}_i$ is an parallel online memory checker if for any family $\{\mathcal{U}_i\}_i$, where each $\mathcal{M}_i$ is an intermediary between $\mathcal{U}_i$ and $\mathcal{S}$ if the following two properties hold:*

- ***Correctness:*** *If $\mathcal{S}$ is honest, then no $\mathcal{M}_i$ aborts and the responses from $\mathcal{M}_i$ to $\mathcal{U}_i$ are all correct with probability* $1 - \mathsf{negl}(\lambda)$.
- ***Soundness:*** *For all p.p.t. $\mathcal{S}$, if* any *$\mathcal{M}_i$ had sent back an incorrect response to $\mathcal{U}_i$, some $\mathcal{M}_j$ (not necessarily the same as $\mathcal{M}_i$) must abort by the end of the last request made by the clients with probability at least* $1 - \mathsf{negl}(\lambda)$.

Concurrency. We sometimes distinguish a family of memory checkers as compatible with EREW, CREW or CRCW PRAM programs. If not explicitly stated, we generally default to the arbitrary CRCW model.

Efficiency metrics. We recap the efficiency metrics as described in Section 1. The main metrics are work and depth blowup, space requirement of the memory checkers, and the server space blowup.

- Depth blowup: The value of ℓ_t (as defined in the first paragraph of this subsection). In other words, this is the ratio of the number of parallel steps conducted by $\{\mathcal{M}_i\}$ for every parallel step of $\{\mathcal{U}_i\}_i$.
- Work blowup: The ratio of $|\vec{I}^{(t,1)} + \vec{I}^{(t,1)} + \cdots + \vec{I}^{(t,\ell_t)}|$ to $\vec{I}^{(t)}$. In other words, is the ratio of the number of physical queries from $\{\mathcal{M}_i\}_i$ to the ratio of underlying logical queries from $\{\mathcal{U}_i\}_i$. We note that we are only charging the communication with $\mathcal{S}$ as work.
- Memory checker local space: This is the amount of secret local space stored by each $\mathcal{M}_i$.
- Server space blowup: This is the size of the server storage divided by Nw (the size of the underlying server storage). In our constructions, this will be $O(1)$ assuming $w = \omega(\log \lambda)$.

5 PRAM Online Memory Checker

In this section, we present our online-memory checking construction achieving $O(\log N)$ blowup in work and depth. Without loss of generality, suppose m and N are powers of two.

Following the construction of Blum et al. [BEG+91], our construction is binary tree of size $2N - 1$, where the leaf nodes correspond to the underlying database.

Algorithm 5.1 Online memory checker for a PRAM with m CPUs sharing a work-tape of size N.

Set-up: A key $\mathsf{sk} \leftarrow \mathsf{MACGen}(1^\lambda)$ is sampled and distributed to all checkers $\{\mathcal{M}_i\}_{i\in[m]}$.
Initial State: The server $\mathcal{S}$'s memory is organized in a binary tree of size $2N - 1$ of height $\log_2 N$. (Note that since the root is the node that is initialized, initialization only takes $O(1)$ time.)

- Initialize the root node r to contain $(r, \mathsf{count} := 0, \mathsf{time} := 0, \mathsf{test} := 0)$ (authenticated).
- Each internal node v is of the form $(v, \mathsf{count}, \mathsf{time})$ (if uninitialized or if authentication fails, treat the contents as $(v, 0, 0)$)[3].

[3]We use this trick to ensure that we obtain worst-case overhead since this mitigates the need for an initialization phase.

- The N leaf nodes of the binary tree correspond to the contents of the logical memory. The leaf node corresponding to addr in the work tape $\boldsymbol{W}$ will contain $(\mathsf{addr}, \mathsf{data}, \mathsf{count}, \mathsf{time}, \mathsf{test})$. In an honest execution, the variables denote the following:
  - addr: Denotes the logical address of the data stored in the leaf.
  - data: Denotes the contents of $\boldsymbol{W}[\mathsf{addr}]$, i.e. the contents of addr in the underlying database.
  - CPU: The CPU which performed the write.
  - count: The number of times $\boldsymbol{W}[\mathsf{addr}]$ was written to.
  - time: The last iteration during which $\boldsymbol{W}[\mathsf{addr}]$ was modified.
  - test : A boolean bit indicating if the last write happened during a "test" phase.

  If uninitialized, treat the contents as $(\mathsf{addr}, \mathsf{data} := \varnothing, \mathsf{CPU} := \varnothing, \mathsf{count} := 0, \mathsf{time} := 0, \mathsf{test} := 0)$.
- Every $\mathcal{M}_i$ has a counter T initialized to 1.

Authentication: Every write is authenticated using $\mathsf{MAC}_{\mathsf{sk}}$ and every read is verified with $\mathsf{Verify}_{\mathsf{sk}}$. If a read fails authentication, we assume that it is an *uninitialized* node.
The algorithm: At iteration T:

- *All readers:* First, every $\mathcal{M}_i$ corresponding to CPUs performing *reads* proceeds as follows.
  - Every $\mathcal{M}_i$ verifies that the root node has $\mathsf{time} = T - 1$ and $\mathsf{test} = 0$. Record the count value at the root.
  - Each $\mathcal{M}_i$ traverses the tree along the path to the leaf associated with addr, in parallel. For each node v along the path with children u and w, verify that $v.\mathsf{count} = u.\mathsf{count} + w.\mathsf{count}$ (i.e. the count values of the corresponding nodes add up). If this is not true for any node, abort and output $\perp$.
  - Once the leaf node is reached, $\mathcal{M}_i$ simply reads the contents of the leaf node corresponding to addr, and sends data to CPU i.
- *All writers:* Now, every $\mathcal{M}_i$ corresponding to CPUs with *writes* proceeds as follows.
  1. *Test phase:* Every $\mathcal{M}_i$ reads the leaf node corresponding to addr. Suppose addr has counter value count. Then, every $\mathcal{M}_i$ tries to write $(\mathsf{addr}, \mathsf{data}', i, \mathsf{count} + 1, \mathsf{time} := T, \mathsf{test} := 1)$ (in parallel) to the leaf node corresponding to addr.
  2. *Winner phase:* Each $\mathcal{M}_i$ reads the same entry to check if their corresponding write had "won" the concurrent write.
     - If yes, rewrite $(\mathsf{addr}, \mathsf{data}', i, \mathsf{count} + 1, \mathsf{time} := T, \mathsf{test} := 0)$ to the same address (i.e. indicate that the test phase has concluded).
     - Otherwise, verify that the count value has been incremented, and $\mathsf{time} = T$, and $\mathsf{test} = 1$.
     - Every $\mathcal{M}_i$ reads the corresponding leaf node again to ensure that it has been updated with the "winning" entry with $\mathsf{test} = 0$ [4].
  3. *Propagation phase:* Now, we propagate the counts from the leaves of the tree to the root of the tree in parallel one layer at a time, starting from the bottom, i.e. updating the nodes from $h = 0, 1, \ldots, \log_2 N$:

[4]If there is a rule for conflict resolution that can be easily verified (e.g. the CPU with the highest priority wins, CPU with the maximum or minimum value write wins, etc.), then that can also be verified here.

  - At $h = 0$ (i.e. leaf nodes), every $\mathcal{M}_i$ with a successful write is assigned to that leaf.
  - At $h \geq 1$, each node with children that are updated will be assigned a checker $\mathcal{M}_i$ as follows:
    * *Assigning nodes*: If the left child was updated at round T, the checker $\mathcal{M}_i$ assigned to the left child is assigned to the node. Otherwise, the checker $\mathcal{M}_i$ assigned to the right child at round T is also assigned to the parent node. Note that this can be checked in an *EREW manner* by simply having a time-step where any checker $\mathcal{M}_i$ assigned to a right node checks if the sibling left node was updated at iteration T. If yes, it will no longer update the values.
  - Every node v at level $h \leq \log_2 N - 1$ with an assigned CPU with children u and w is updated to $(v, u.\mathsf{count} + w.\mathsf{count}, T)$. At $h = \log_2 N$ (i.e. the root node), the root r is set to $(r, u.\mathsf{count} + w.\mathsf{count}, T, \mathsf{test} := 1)$.

4. *Verification of the root:*
  - In a separate array of size m (i.e. the number of CPUs), every $\mathcal{M}_i$ writes a 1 if it performed a successful write operation (i.e. "won" in the "winner phase") to the database, and 0 otherwise.
  - Compute the sum of this array to be some W. Note that this can be done in $O(\log m)$ depth with $O(m)$ work with an EREW algorithm.
  - $\mathcal{M}_1$ verifies that the count value of the root of the tree was increased by W (note that W can be 0 if no writes were performed).
    * If the count count was in fact correct, update the root to be $(\mathsf{addr}, \mathsf{count}, T, 0)$.
    * Otherwise, abort and output $\perp$.

- Every $\mathcal{M}_i$ locally increments T.

First, we make a couple of remarks about Algorithm 5.1.

Remark 5.2. *If the underlying algorithm is CREW, the access patterns of the memory checkers are also CREW if the server is honest. If the underlying algorithm is EREW, we can make the following modifications to ensure the memory checkers' access patterns are also EREW. One can treat every "read" also as a "write" where the same value is written back. Then, we can simply skip "All readers" phase of the above algorithm and execute the "All writers" phase. Since exactly one user is assigned to each internal node when the server is honest, the resulting memory checking algorithm is also EREW against honest servers.*

Remark 5.3. *We can relax the parallelization requirements of the model by instead having an agreed upon time for "read rounds" and "write rounds" for the "All readers" and "All writers" phase. This ensures that all CPUs can agree on the time-stamp of the root of the tree. We can also assume that in practice, no writes happen concurrently (e.g. by adding randomness to the timing of an access), and the algorithm is essentially EREW. We also assume that any read followed by an immediate write to the same location is "atomic" and cannot be interfered (i.e. any read-write to update the counter will not be interleaved with another CPU's read-write, as this will result in inconsistent counters). The "read rounds" have no synchronization issues, as long as every CPU agrees on the counter at the root of the tree. During the "write rounds", the protocol is consistent during leaf-update phase and propagation phases as long as read-writes are atomic. During the verification phase, we perform an EREW algorithm sum a set of values and compare it against the root. As long as this is done in a consistent and authenticated way, this phase can be done correctly and securely.*

Theorem 5.4. *Assuming the existence of one-way functions, there exists an online memory checker for a m-CPU PRAM with $O(\log N)$ work blowup and $O(\log N)$ depth blowup. Each CPU locally needs to store $O(1)$ words and one PRF key of length $\ell(\lambda)$ (for authentication).*

Proof. We first start by mentioning some invariants preserved during the algorithm when the server is honest.

Invariants. For each node u in the tree, we denote by $u.\mathsf{count}(t)$ the value the count associated to u that is read at iteration t. The following invariants are maintained if the server behaves honestly at the start of iteration T.

- Every $\mathcal{M}_i$ has the correct value T of iteration number.
- Every entry is authenticated, unless uninitialized (we treat every entry which fails authentication as uninitialized). An uninitialized node v is treated to have $v.\mathsf{count}(t) = 0$.
- The time value of the root node is equal to $T - 1$.
- For a leaf node v corresponding to logical address addr, the variable $v.\mathsf{count}(t)$ denotes the number of times the logical address addr is written to. Additionally, $v.\mathsf{time}$ is the last iteration when any of the leaves of the sub-tree is updated.
- For every non-leaf node v, the variable $v.\mathsf{count}(t)$ denotes the *total* number of times the leaf nodes of the sub-tree rooted at v have been updated. In particular, if v and w are the children of u, we have that
$$u.\mathsf{count}(t) = v.\mathsf{count}(t) + w.\mathsf{count}(t).$$
- For every node v, the variable $v.\mathsf{time}$ is the last iteration when any of the leaves of the sub-tree was updated.

Clearly, the invariants are met at the initialization phase of the algorithm.

Suppose that we are at iteration T of the algorithm, and that the server has functioned honestly so far (up to iteration $T - 1$) and the invariants have been maintained. Now, we argue that at the end of iteration T, either the memory functions correct correctly and the invariants are preserved, or the memory functions incorrectly and some $\mathcal{M}_i$ aborts and outputs $\perp$.

The former case is easy. Therefore, it suffices to consider the case where the memory functions incorrectly. By the inductive hypothesis, note that $u.\mathsf{count}(t)$ for $t \leq T - 1$ are correct for every node u in the binary tree.

By unforgeability of MACs, every valid read corresponds to some authenticated write with probability $1 - \mathsf{negl}(\lambda)$. Therefore, for the rest of this proof, we limit the memory's attacks to only *replay attacks* (i.e. memory sends stale requests corresponding to each address). There are three cases to consider when the memory functions incorrectly.

Case 1: Memory functions incorrectly during the traversal phase of the reads.

First, notice that the time value of the root is monotone increasing with each iteration. Therefore any replay attack on the root node will show $\mathsf{time} < T$, and will be detected. For every internal node and leaf node, notice that the count value is monotone increasing with every write if the memory functions correctly. Suppose the memory first performs a replay attack on node u (not equal to the root) while some $\mathcal{M}_i$ is traversing a path from the root to some leaf. Then, we have $v.\mathsf{count}(T) > u.\mathsf{count}(T) + w.\mathsf{count}(T)$ where v is the parent node of u, and w is a sibling node of u. Therefore, any such replay attack will be caught and $\mathcal{M}_i$ will immediately abort.

Case 2: Memory functions incorrectly during the test and/or winner phases.

Let W be the number of CPUs that "win" during the test phase, and let U be the number of leaf nodes that are updated with writes. Note that it is possible that multiple CPUs get the signal that they won the concurrent write. Let $w_1, w_2, \ldots, w_U$ denote the number of "wins" associated to each of the U nodes that are updated with writes, i.e. number of CPUs that get a signal that they won per updated leaf node. Since every $\mathcal{M}_i$ with a write verifies that some CPU has won that concurrent write (with $\mathsf{test} = 0$), this ensures that $w_j \geq 1$. Moreover, $w_j = 1$ if and only if there is exactly one CPU that won the arbitrary write.

Therefore, we have that $W = \sum_{j=1}^{U} w_j \geq U$, where equality holds if and only if exactly one CPU wins each concurrent write. In other words, if the memory functions incorrectly in this phase, we must have $W > U$. Recall that W is computed during the verification phase, and we argue in the next case that this inequality will be detected during the verification phase.

Case 3: Memory functions incorrectly during the propagation and/or the verification phases.

Consider an arbitrary internal node v of the tree. Let $v.\mathsf{count}(T-1)$ be the count value at iteration $T-1$ (this is correct by the induction hypothesis), and let $v.\mathsf{update}$ be the number of leaf nodes of the sub-tree at v that were updated at iteration T (possibly by multiple nodes). We argue by induction on the height of v that for any read to v, the counter value read is at most $v.\mathsf{count}(T-1)+v.\mathsf{update}$. Moreover, equality holds if and only if the memory functioned correctly in the sub-tree rooted at v during the propagation phase.

At $h = 0$ (i.e. leaf nodes), the statement is clearly true because a replay attack can only show a smaller counter value than $v.\mathsf{count}(T-1)$ by the monotonicity of the count values. Therefore, any read to v can only show a value of at most $v.\mathsf{count}(T-1)$ if it was not updated, and $v.\mathsf{count}(T-1)+1$ otherwise.

Now, consider a node v at height $h' \geq 1$. Suppose that u and w are the children of v, and that the node updating v receives counter values $u.\mathsf{count}(T)$ and $w.\mathsf{count}(T)$ when reading u and w respectively. Then, the new updated counter value of v is

$$\begin{aligned} v.\mathsf{count}(T) &= u.\mathsf{count}(T) + v.\mathsf{count}(T) \\ &\leq u.\mathsf{count}(T-1) + u.\mathsf{update} + w.\mathsf{count}(T-1) + w.\mathsf{update} \\ &= v.\mathsf{count}(T-1) + v.\mathsf{update} \end{aligned}$$

where the first equality holds by the definition of $v.\mathsf{count}(T)$, the second inequality holds by the induction hypothesis, and the last equality comes from the correctness of the memory at iteration $T-1$, and by the definition of $v.\mathsf{update}$. Therefore, the largest possible count value associated to any read of v in this iteration is at most $v.\mathsf{count}(T-1)+v.\mathsf{update}$, where equality holds if and only if the memory functioned correctly in the sub-tree rooted at v during the propagation phase.

In particular, at the root node r, we have that

$$r.\mathsf{count}(T) \le r.\mathsf{count}(T-1) + r.\mathsf{update} \le r.\mathsf{count}(T-1) + U$$

where equality holds if and only if the memory functioned correctly during the propagation phase. Moreover, combining this with Case 2, we have that

$$r.\mathsf{count}(T) - r.\mathsf{count}(T-1) \le U \le W$$

where all the inequalities hold if and only if the memory functioned correctly at every point in the algorithm. Therefore, if the memory functions incorrectly at any point, the check at the verification phase will fail, thereby completing the proof.

Efficiency. Note that each $\mathcal{M}_i$ traverses a path to the desired leaf of CPU i, and therefore does a $O(\log N)$ depth traversal. Moreover, during the propagation phase, each CPU again updates only elements on the path from the root to its leaf of the tree, and hence will once again only update $O(\log N)$ elements.

Moreover, it is clear that each checker only requires $O(1)$ local space to keep track of the root of the tree as well as verify the count values of the nodes of the tree. Therefore, this gives us the desired and space complexity. □

6 Statistically Secure EREW PRAM Offline Memory Checker

In this section, we show that the offline memory checking approach of Blum et al. [BEG+91] can be naturally parallelized and extended to the EREW PRAM setting. We state the theorem here.

Definition 6.1. *A family of hash functions $\mathcal{H} = \{h_k\}_{k\in\mathcal{K}}$ where $h_k : \{0,1\}^n \to \{0,1\}^{\ell(n)}$ is ϵ-biased if*

$$\Pr_k[h_k(x) = h_k(y)] \le \epsilon.$$

Naor and Naor [NN90] constructed a family of efficient ϵ-biased hash functions as follows.

Theorem 6.2 ([NN90])**.** *There exists a family of ϵ-biased hash functions $\mathcal{H} = \{h_k\}_k$ with $h_k : \{0,1\}^n \to \{0,1\}^{\ell(n)}$ such that $\ell(n) = O\left(\log n + \log \frac{1}{\epsilon}\right)$ and $|k| = O\left(\log n + \log \frac{1}{\epsilon}\right)$.*

Moreover, the family of hash functions of [NN90] has the additional property that it is in fact *linear*. In other words, h_k is of the form $h_k(x) = \boldsymbol{A}x \pmod 2$ for a matrix $\boldsymbol{A} \in \{0,1\}^{\ell(n)\times n}$, where the ith column $\boldsymbol{A}_i$ of $\boldsymbol{A}$ can be computed efficiently given k. Therefore, $h_k(x)$ can be computed incrementally given the bits of x.

In this construction, every memory location on the server $\mathcal{S}$ contains the following information:

- addr: Logical address

- data: Contents of logical address addr.
- CPU: Name of CPU that last accessed (could be either read or write).
- time: Last time CPU accessed addr.

For an EREW algorithm of depth d, the values written to any memory location must be in the set $Z = [N] \times \{0,1\}^w \times [m] \times [d]$, and any such value is written at most once. Therefore, there are $z = |Z| = N \times 2^w \times m \times d = O(\mathsf{poly}(N))$ possible values of address.

Following Blum et al. [BEG+91], let R be the set of values that are read by the CPUs, and let W be the set of values that are written by the CPUs. Note that we can encode $R, W \subseteq Z$ as indicator vectors in $\{0,1\}^z$. Our algorithm will keep track of $h(R)$ and $h(W)$.

Without loss of generality, we may assume m divides N.

Algorithm 6.3 Efficient EREW Parallel Memory Checker

Set-up phase:

- Pick $\epsilon > 0$ such that $\epsilon = \mathsf{negl}(\lambda)$.
- Fix a family of ϵ-biased hash functions $\{h_k\}_{k \in \mathcal{K}}$ such that $h_k : \{0,1\}^z \to \{0,1\}^{\ell(z)}$. Sample $k \leftarrow \mathcal{K}$, and distribute k to all CPUs.
- For each $\mathcal{M}_i$, sample two random strings $r_i^{(1)}, r_i^{(2)} \leftarrow \{0,1\}^{\ell(z)}$. Let $\boldsymbol{A}$ represent the matrix corresponding to h_k. Associate each column of $\boldsymbol{A}$ with some value in $Z := [N] \times \{0,1\}^w \times [m] \times [d]$.
- $\mathcal{M}_1$ additionally receives $r^{(j)} = \oplus_{i=1}^m r_i^{(j)}$ for $j = 1, 2$.
- Every $\mathcal{M}_i$ has a counter T initialized to 1.
- Every $\mathcal{M}_i$ keeps track of a value h_i^R and h_i^W of length $\ell(z)$, intialized to $0^{\ell(z)}$.
- We abuse the notation $\mathcal{S}[\mathsf{addr}]$ to denote the underlying database entry at address addr along with the last CPU CPU which updated the entry, and the last time-step time when the entry was updated. We also have additional $O(N)$ server space for the second phase of the algorithm.

Query Phase:

- Initialize all server memory locations to 0^w.
- At iteration T:
  - Every $\mathcal{M}_i$ corresponding to a CPU with reqeust $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ does the following in parallel:
    - Retrieve $\mathcal{S}[\mathsf{addr}]$ to obtain $(\mathsf{addr}_{\mathsf{old}}, \mathsf{data}_{\mathsf{old}}, \mathsf{CPU}_{\mathsf{old}}, \mathsf{time}_{\mathsf{old}})$. We denote this string by $x \in Z$.
    - Verify that $\mathsf{time}_{\mathsf{old}} < T$ and $\mathsf{addr}_{\mathsf{old}} = \mathsf{addr}$.
    - Update $h_i^R \leftarrow h_i^R + \boldsymbol{A}_x$, where $\boldsymbol{A}_x$ is the column of $\boldsymbol{A}$ corresponding to x.
    - If $\mathsf{op} = \mathsf{read}$, set $\mathsf{data} = \mathsf{data}_{\mathsf{old}}$.
    - Update $\mathcal{S}[\mathsf{addr}]$ to be $(\mathsf{addr}, \mathsf{data}, i, T)$. We denote this string by $y \in Z$.
    - Update $h_i^W \leftarrow h_i^W + \boldsymbol{A}_y$.
  - Every $\mathcal{M}_i$ increments T locally.

Verification phase:

- Now, every $\mathcal{M}_i$ in parallel reads memory locations $\mathsf{addr} \in [N/m \cdot (i-1)+1, N/m \cdot i]$ and does the following:
  - Retrieve $x \leftarrow \mathcal{S}[\mathsf{addr}]$.
  - Update $h_i^R \leftarrow h_i^R + \boldsymbol{A}_x$.
- First, we compute $h(W) = \sum_{i=1}^m h_i^W$ as follows (using server space):
  - Every $\mathcal{M}_i$ posts $h_i^W \oplus r_i^{(1)}$ on the additional work space $\mathcal{S}$ (i.e. a one-time pad of the secret state).
  - Compute $\sum_{i=1}^m (h_i^W \oplus r_i^{(1)}) = r^{(1)} \oplus h(W)$ in a tree-like manner (note that this is EREW).
  - $\mathcal{M}_1$ locally computes $r^{(1)} \oplus r^{(1)} \oplus h(W) = h(W)$ and stores this value locally.
- Now, we similarly compute $h(R) = \sum_{i=1}^m h_i^R$ as follows:
  - Every $\mathcal{M}_i$ posts $h(R) = h_i^R \oplus r_i^{(2)}$ on the additional work space of $\mathcal{S}$.
  - Compute $\sum_{i=1}^m (h_i^R \oplus r_i^{(2)}) = r^{(2)} \oplus h(R)$ in a tree-like manner.
  - $\mathcal{M}_1$ locally computes $r^{(2)} \oplus r^{(2)} \oplus h(R) = h(R)$ and stores this value locally.
- $\mathcal{M}_1$ only accepts if $h(R) = h(W)$, and rejects otherwise.

Comparison to Blum et al. [BEG$^+$91]. If $m = 1$ (i.e. we are in the RAM setting), note that the CPU can aggregate the value of the hash function locally. Therefore, the server $\mathcal{S}$ does not see evaluations of h_k and hence has no information about k. However, when $m > 1$, if the users aggregate the values of the hash function in the clear using the server space, the server might be able to determine information about k. Therefore, it seems like encryption might be necessary. However, since each server only has to encrypt two messages in our scheme, we can instead rely on the information-theoretically secure one-time pad.

Theorem 6.4. *Consider $w = \Theta(\log N)$. Consider an EREW algorithm with work q and depth d. There is a statistically secure offline memory checker for a m-CPU EREW algorithm with total work $O(q + N + m)$ and depth $O(d + N/m + \log m)$. Each $\mathcal{M}_i$ has to locally store $O(\log N + \log(1/\epsilon))$ bits of memory, where $\epsilon = \mathsf{negl}(\lambda)$.*

Proof. Following the argument of Blum et al. [BEG$^+$91, Lemma 1], we have that $R = W$ if and only if the server did not corrupt any of the responses to the memory checkers. Therefore, using the epsilon-biased hash, we have that $h(R) = \sum_{i=1}^m h_i^R$ and $h(W) = \sum_{i=1}^m h_i^W$ are not equal with probability at least $1 - \epsilon$ if the server responded incorrectly.

Now, it suffices to show that $\mathcal{S}$ cannot manipulate the verification phase. Note that since the server only sees one-time pads of the values $h_i^R \oplus r_i^{(1)}$ and $h_i^W \oplus r_i^{(2)}$ and does not see the values of $r^{(1)}$ and $r^{(2)}$, the server only sees truly random values. Therefore, even if it does manipulate the values sent back to the servers, the probability that $\mathcal{M}_1$ accepts is still at most ϵ. □

Corollary 6.5. *Consider $w = \Theta(\log N)$. Consider an arbitrary CRCW algorithm algorithm with work q and depth d. There is a statistically secure offline memory checker for a m-CPU EREW algorithm with total work $O(q + md\log m + N + m)$ and depth $O(d\log m + N/m + \log m)$. Each $\mathcal{M}_i$ has to locally store $O(\log N + \log(1/\epsilon))$ bits of memory, where $\epsilon = \mathsf{negl}(\lambda)$.*

Proof. Using Lemma 3.1, we have that the algorithm can be converted into an EREW algorithm with work $q + md \log m$ and depth $d \log m$. Now, we get our result by applying Theorem 6.4. □

We note that this yields a CRCW offline memory checking algorithm with $O(\log m)$ amortized blow-up in work and depth. In the next section, we show how to achieve $O(1)$ amo

7 PRAM Offline Memory Checker

In this section, we use authentication to construct an offline memory checker for CREW and CRCW PRAM programs with amortized $O(1)$ complexity in both work and depth. Throughout this section, we are once again in the setting where the word size is $w = \omega(\log \lambda)$. Without loss of generality, we may assume that m divides N, where m is the number of CPUs and N is the size of the database.

Algorithm 7.1 Offline memory checker for the arbitrary CRCW model

Set-up phase:

- Fix a MAC family $(\mathsf{MACGen}, \mathsf{MAC}, \mathsf{Verify})$. Sample $\mathsf{sk} \leftarrow \mathsf{MACGen}(1^\lambda)$ and distribute sk to all $\mathcal{M}_i$.
- Each $\mathcal{M}_i$ associated to CPU i initializes a local counter t_i to 0.
- We abuse the notation $\mathcal{S}[\mathsf{addr}]$ to denote the underlying database entry data at address addr along with the following metadata $\mathbf{MD}$ containing $(\mathsf{count}, \mathsf{CPU}, \mathsf{time}, \mathsf{CPU}_{\mathsf{prev}}, \mathsf{time}_{\mathsf{prev}}, \mathsf{test})$. In an honest execution, these entries correspond to the following:
  - count: Number of times this logical address accessed.
  - CPU: Name of CPU that last accessed (could be either read or write).
  - time: Last time CPU accessed addr.
  - $\mathsf{CPU}_{\mathsf{prev}}$: Name of CPU that accessed addr before time.
  - $\mathsf{time}_{\mathsf{prev}}$: Last time $\mathsf{CPU}_{\mathsf{prev}}$ accessed addr.
  - test: A boolean bit indicating if the last write happened during the "test" phase.
- We also have additional $O(N)$ server space for the second phase of the algorithm.

Authentication: Every write is authenticated using $\mathsf{MAC}_{\mathsf{sk}}$ and every read is verfied with $\mathsf{Verify}_{\mathsf{sk}}$. If a read fails authentication, we abort.

Query phase:

- Initialize $\mathcal{S}[\mathsf{addr}]$ to set $\mathsf{addr} := \mathsf{addr}$, $\mathsf{count} := 0$, $\mathsf{test} := 0$, and set all other fields metadata fields $\mathsf{CPU}, \mathsf{time}, \mathsf{CPU}_{\mathsf{prev}}$, $\mathsf{time}_{\mathsf{prev}}$ to $\varnothing$.
- For each batch of requests $\vec{I}^{(T)} = (I_i^{(T)} : i \subseteq [m])$ at time T:
  - *All readers:* First, we handle all requests $I_i = (\mathsf{op}_i, \mathsf{addr}_i, \mathsf{data}_i)$ from which are *reads.*
    - ∗ *Read phase:* Each $\mathcal{M}_i$ corresponding to $\mathsf{op}_i = \mathsf{read}$ does the following in parallel:
      - · Read the corresponding entry $\mathcal{S}[\mathsf{addr}_i]$, and downloads and saves the contents $(\mathsf{addr}, \mathsf{data}, \mathbf{MD})$. Abort if $\mathsf{addr} \neq \mathsf{addr}_i$, or if the test value is nonzero.
      - · Return data to CPU i.
      - · Unpack $\mathbf{MD} = (\mathsf{count}, \mathsf{CPU}, \mathsf{time}, \mathsf{CPU}_{\mathsf{prev}}, \mathsf{time}_{\mathsf{prev}}, \mathsf{test})$.
      - · Set $\mathsf{CPU}'_{\mathsf{prev}} := \mathsf{CPU}$ and $\mathsf{time}'_{\mathsf{prev}} = \mathsf{time}$.

        · Set $\mathsf{CPU}' := i$ and $\mathsf{time}' := T$.
        · Increment $\mathsf{count}' := \mathsf{count} + 1$.
        · Set $\mathsf{test}' := 1$.
        · Set $\mathbf{MD}' = (\mathsf{count}', \mathsf{CPU}', \mathsf{time}', \mathsf{CPU}'_{\mathsf{prev}}, \mathsf{time}'_{\mathsf{prev}}, \mathsf{test}')$.
    - *Test phase:* All $\mathcal{M}_i$'s perform a write to the accessed address with $(\mathsf{addr}_i, \mathsf{data}, \mathbf{MD}')$.
    - *Winner phase:* Each $\mathcal{M}_i$ performs a read to see if it "won" the arbitrary write. Note that when the memory is honest, there should be exactly one write that wins.
        · If yes, write back to update the test value to be 0. Increment local counter t_i.
        · Else, verify that $\mathsf{CPU}'_{\mathsf{prev}}$, $\mathsf{time}'_{\mathsf{prev}}$ and count' values of the winning write are consistent with their own write attempt, and that test value is updated to 0. [5]
    - Every $\mathcal{M}_i$ increments their locally stored global timer T.
  – *All writers:* Same as read phase, except data is now updated with data_i during the test and winner phases.

Verification phase:

- Compute $t := \sum_i t_i$, i.e. the sum of all local counters of all the CPUs.
- Denote $c_{\mathsf{addr}} := S[\mathsf{addr}].\mathsf{count}$. Compute $t' := \sum_{\mathsf{addr}} c_{\mathsf{addr}}$ in a tree-like manner.
- Accept if and only if $t = t'$, otherwise abort and output $\perp$.

Remark 7.2. *As in the online memory checking case, we can relax the parallelization requirements of the model for this memory checking protocol. During the query phase, we assume that in practice, no reads or writes happen concurrently (e.g. by adding randomness to the timing of an access), and the algorithm is essentially EREW. We also assume that any read followed by an immediate write to the same location is "atomic" and cannot be interfered (i.e. any read-write to update the counter will not be interleaved with another CPUs read-write, as this will result in inconsistent counters). For the verification phase, every CPU needs to agree when the verification phase begins, and when to write their respective local counters on the server. After this, the CPUs have to sum two lists of numbers and compare them. Since summing a list is an EREW algorithm, as long as this is done in a consistent and authenticated way, we can ensure the security and correctness of our protocol.*

Theorem 7.3. *Consider an honest-but-curious implementation with work q and depth d. Then, there is a post-verifiable offline memory checker with total work $O(q+N)$, total depth $O(d+N/m+\log m)$ and local space complexity $O(1)$ words and one PRF key of length $\ell(\lambda)$.*

Proof. By unforgeability of MACs, every valid read corresponds to some authenticated write with probability $1 - \mathsf{negl}(\lambda)$. Moreover, since we check the consistency of addr when we read, for the rest of this proof, we limit the memory's attacks to only *replay attacks* (i.e. memory sends stale requests corresponding to each address).

History graph. For each address $\mathsf{addr} \in \mathcal{S}$, let V_{addr} be the set of $(\mathsf{count}, \mathsf{CPU}, \mathsf{time})$ tuples corresponding to contents written to addr with flag $\mathsf{test} = 0$. Construct the following directed acyclic graph G_{addr} on V_{addr}.

[5] If there is a rule for conflict resolution that can be easily verified (e.g. the CPU with the lowest number wins), then that can also be verified here.

- The root of the graph is $(0, \varnothing, 0)$.
- If something of the form $(\mathsf{addr}, *, \mathsf{count}, \mathsf{CPU}, \mathsf{time}, \mathsf{CPU}_{\mathsf{prev}}, \mathsf{time}_{\mathsf{prev}}, \mathsf{test} := 0)$ was ever written to addr, add an edge from $(\mathsf{count} - 1, \mathsf{CPU}_{\mathsf{prev}}, \mathsf{time}_{\mathsf{prev}})$ to $(\mathsf{count}, \mathsf{CPU}, \mathsf{time})$. Here, * denotes that the data entry can be any arbitrary value.

Intuitively, this graph will represent the *history* of updates made to addr. We abuse the notation $|V_{\mathsf{addr}}|$ to denote the number of vertices in G_{addr}.

Claim 7.4. *If the server behaved correctly, then* $c_{\mathsf{addr}} = |V_{\mathsf{addr}}| - 1$. *Otherwise, if the server behaved incorrectly, for some* $\mathsf{addr} \in \mathcal{S}$, *we must have* $c_{\mathsf{addr}} < |V_{\mathsf{addr}}| - 1$.

Proof. First, we argue that the history graph G_{addr} for each $\mathsf{addr} \in \mathcal{S}$ is connected. For all nodes with count value 1, it must be adjacent to the root node $(0, \varnothing, 0)$. Therefore, all the nodes with count at most 1 are connected. Now assume that all nodes with count at most $k - 1$ are connected. Note that a node with $\mathsf{count} = k$, it must be adjacent to some node with $\mathsf{count} = k - 1$, and hence inductively, we must have that G_{addr} is connected.

Let h_{addr} denote the height of the graph G_{addr}. It is clear that c_{addr} must correspond to the count value of some node in the graph, and hence $c_{\mathsf{addr}} \leq h_{\mathsf{addr}}$. Note that $h_{\mathsf{addr}} = |V_{\mathsf{addr}}| - 1$ if and only if G_{addr} is a path. Hence, to show the statement of the theorem, it suffices to show that G_{addr} is a path if and only if the server behaves correctly.

Clearly, if the server behaves correctly, for every $\mathsf{addr} \in \mathcal{S}$, the graph G_{addr} is a path. Moreover, the final version of the $\mathcal{S}[\mathsf{addr}]$ corresponds to the leaf node of G_{addr}, and hence has count $|V_{\mathsf{addr}}| - 1$, as desired.

Now, suppose the server behaves incorrectly at some address addr. There are three ways that the memory could have functioned incorrectly.

- The memory functioned correctly throughout, until the final read to $\mathcal{S}[\mathsf{addr}]$ during the verification phase, where the memory does a replay attack. Then, the graph is a path, but $c_{\mathsf{addr}} < |V_{\mathsf{addr}}| - 1$ since the memory must have sent back a counter associated to a non-leaf node.
- The memory could have sent back a "stale" entry for some address addr. Consider the first such stale response. Note that because of the $\mathsf{test} = 0$ flag check, the "stale" request must correspond to some node on G_{addr}, say $(\mathsf{count}, \mathsf{CPU}, \mathsf{time})$. Moreover, since this is a stale request, it must mean that this node already has a child in G_{addr}. Note that the new update created must have the form $(\mathsf{addr}, *, \mathsf{count} + 1, \mathsf{CPU}', \mathsf{time}', \mathsf{CPU}, \mathsf{time})$. Therefore, $(\mathsf{count}, \mathsf{CPU}, \mathsf{time})$ has at least two children, and the graph is no longer a path.
- Alternatively, the memory could have accepted conflicting writes to some addr at the same time-step. In particular, some $\mathcal{M}_i$ and $\mathcal{M}_j$ concurrently write to some addr at time-step time, and both writes "win" in the winner phase. Then, the corresponding winning writes, $(\mathsf{count}_i, \mathsf{CPU}_i, \mathsf{time})$ and $(\mathsf{count}_j, \mathsf{CPU}_j, \mathsf{time})$, cannot lie on the same path from the root because they both have the same time value (because by construction, the time values are increasing on any directed path from the root).

□

If the server behaves correctly, note that $t = \sum t_i = \sum |V_{\mathsf{addr}}| - 1$ because this is the number of times each address is updated, and $t' = \sum_{\mathsf{addr}} c_{\mathsf{addr}} = \sum |V_{\mathsf{addr}}| - 1 = t$. Therefore, the memory checker accepts.

Otherwise, if the server behaves incorrectly, then for some address addr', $c_{\mathsf{addr}'} \leq |G_{\mathsf{addr}'}| - 2$. Therefore, $T' \leq \sum_{\mathsf{addr} \neq \mathsf{addr}'}(|V_{\mathsf{addr}}| - 1) + (|G_{\mathsf{addr}'}| - 2) < \sum(|V_{\mathsf{addr}}| - 1) = \sum t_i = T$, and hence will be rejected with probability $1 - \mathsf{negl}(\lambda)$.

Efficiency. During the query phase, it is easy to see that every underlying query generates $O(1)$ physical queries. Therefore, the work during the query phase is $O(q)$, and the depth is $O(d)$. During the second phase of the algorithm, we are summing $O(N)$ counters, and this takes $O(N + m)$ work and $O(N/m + \log m)$ parallel steps. This gives us the desired efficiency. $\square$

8 Maliciously Secure Oblivious Parallel RAM

In this section, we recall the definition of oblivious parallel RAMs (OPRAMs), as introduced by Boyle et al. [BCP16]. We then extend the notion of OPRAMs to be secure even in the presence of malicious adversaries. For a full treatment of malicious security of oblivious RAMs, we refer the readers to [MV23]. Then, we argue that composing our memory checker with any honest-but-curious oblivious PRAM gives a maliciously secure OPRAM construction. As a corollary, using the honest-but-curious construction of [AKL+22], we obtain an OPRAM with $O(\log^2 N)$ simulation overhead in both work and depth for a database of size N.

8.1 Definitions

Reactive PRAM functionalities. Loosely speaking, a reactive functionality $\mathcal{F}$ is an interactive functionality that holds some internal state, and whenever it takes in a command cmd and input x, it gives some (possibly randomized) output $\mathcal{F}(\mathsf{cmd}, x)$, where the notational dependence on the internal hidden state is suppressed. One way to think of a reactive functionality is as a specification for a data structure problem (i.e., the desired behavior of a data structure), where the various types of queries are specified by cmd and the input to those queries are denoted by x. For a PRAM reactive functionality for a m-processor machine, we can instead model it as taking as input $I = \{(\mathsf{cmd}_i, x_i\})_{i \in [m]}$, and outputting a list $\{\mathsf{out}\}_{i \in [m]}$, where out_i is sent to CPU i. Note that it is possible that only a subset of the CPUs make a request at any point in time, and we set $(\mathsf{cmd}_i, x_i) = (\varnothing, \perp)$ if CPU i does not make a request. We can write the PRAM model itself as a reactive PRAM functionality as described in $\mathcal{F}^{N,w}_{\mathsf{PRAM}}$ (Functionality 8.1). We use the notation $\widehat{\mathsf{cmd}}, \widehat{x}$ and $\widehat{\mathsf{out}}$ to denote the list of all commands $\{\mathsf{cmd}_i\}_{i \in [m]}$, list of all inputs $\{x_i\}_{i \in [m]}$ and list of all outputs $\{\mathsf{out}_i\}_{i \in [m]}$ respectively.

Oblivious implementations. Given an m-CPU PRAM reactive functionality $\mathcal{F}$, we say that an *oblivious implementation* $\Pi = \{\mathcal{C}_i\}_{i \in [m]}$ (each $\mathcal{C}_i$ is a compiler acting on CPU i) is a PRAM compiler that interacts with the memory and has identical input/output behavior as $\mathcal{F}$, and additionally has a simulatable access pattern. In more detail, we consider an adversary $\mathcal{A}$ that participates in

Functionality 8.1 $\mathcal{F}^{N,m}_{\mathsf{PRAM}}$: The Arbitrary CRCW PRAM Functionality.

Syntax:

- $\mathsf{cmd}_i \in \{\mathsf{read}, \mathsf{write}, \varnothing\}$, where $\varnothing$ indicates that CPU i is not reading or writing.
- $x_i = (\mathsf{addr}_i, \mathsf{data}_i) \in [N] \times \{0,1\}^w$, where addr_i is an index into an N-entry RAM database, and data_i contains a word of size w(to be used only if $\mathsf{cmd}_i = \mathsf{write}$).

Internal State: A memory array mem with N entries, each containing values in $\{0,1\}^w$, all initialized to 0^w.

Command $\mathcal{F}^{N,m}_{\mathsf{PRAM}}((\mathsf{cmd}_1, x_1), (\mathsf{cmd}_2, x_2), \ldots, (\mathsf{cmd}_m, x_m))$:

- If $\mathsf{cmd}_i = \mathsf{read}$, set $\mathsf{out}_i = \mathsf{mem}[x_i[\mathsf{addr}]]$.
- If $\mathsf{cmd}_i = \mathsf{write}$, update mem by $\mathsf{mem}[x_i[\mathsf{addr}]] \leftarrow x_i[\mathsf{data}]$, and a success symbol is sent back to the user (i.e., only the internal hidden state is modified). If there are multiple users accessing the same addr, an arbitrary user's update is chosen. The remaining users get a $\perp$ symbol.

Experiment 8.2 or Experiment 8.2. We use the shorthand $(\widehat{\mathsf{out}}, \mathsf{Addrs}) \leftarrow \Pi(\widehat{\mathsf{cmd}}, \widehat{x})$ denote the joint distribution of Π's outputs and access patterns. In the real experiment, $\mathcal{A}$ sees this joint distribution produced by Π. In the ideal experiment, it receives an access pattern Addrs generated by a simulator $\mathcal{S}$, and independently generated functionality output. Furthermore, $\mathcal{A}$ is allowed to choose the next command and input in an adaptive manner in either experiment. We formalize this in Definition 8.4.

Experiment 8.2 $\textsc{Real}^{\mathsf{HBC}}(\Pi, \mathcal{A})$.

```
$(\widehat{\mathsf{cmd}}, \widehat{x}) \leftarrow \mathcal{A}(1^\lambda)$
while $\widehat{\mathsf{cmd}} \neq \perp$ do
    $\boxed{\widehat{\mathsf{out}}, \mathsf{Addrs} \leftarrow \Pi(1^\lambda, \widehat{\mathsf{cmd}}, \widehat{x})}$
    $(\widehat{\mathsf{cmd}}, \widehat{x}) \leftarrow \mathcal{A}\left(1^\lambda, \boxed{\widehat{\mathsf{out}}}, \mathsf{Addrs}\right)$
end while
return $b \leftarrow \mathcal{A}(1^\lambda)$
```

Experiment 8.3 $\textsc{Ideal}^{\mathsf{HBC}}(\mathcal{F}, \mathcal{S}, \mathcal{A})$.

```
$(\widehat{\mathsf{cmd}}, \widehat{x}) \leftarrow \mathcal{A}(1^\lambda)$
while $\widehat{\mathsf{cmd}} \neq \perp$ do
    $\boxed{\mathsf{Addrs} \leftarrow \mathcal{S}(1^\lambda, \widehat{\mathsf{cmd}})}$
    $(\widehat{\mathsf{cmd}}, \widehat{x}) \leftarrow \mathcal{A}\left(1^\lambda, \boxed{\mathcal{F}(\widehat{\mathsf{cmd}}, \widehat{x})}, \mathsf{Addrs}\right)$
end while
return $b \leftarrow \mathcal{A}(1^\lambda)$
```

Definition 8.4 (Oblivious implementation)**.** *For a reactive functionality $\mathcal{F}$, we say a (stateful) PRAM machine $\Pi_{\mathcal{F}} = \{\mathcal{C}_i\}_{i\in[m]}$ is a $(1-\delta)$-oblivious implementation of a reactive functionality $\mathcal{F}$ if there is a (stateful) PPT simulator $\mathcal{S}$ such that for all (stateful) PPT $\mathcal{A}$, the adversary $\mathcal{A}$ distinguishes between $\textsc{Real}^{\mathsf{HBC}}(\Pi_{\mathcal{F}}, \mathcal{A})$ (Experiment 8.2) and $\textsc{Ideal}^{\mathsf{HBC}}(\mathcal{F}, \mathcal{S}, \mathcal{A})$ (Experiment 8.3) with advantage at most δ.*

We can view the honest-but-curious OPRAM definition of Boyle, Chung and Pass as an oblivious implementation Π of the $\mathcal{F}^{N}_{\mathsf{PRAM}}$ reactive functionality (Functionality 8.1).

Definition 8.5 (OPRAM)**.** *We say that Π is an* honest-but-curious oblivious PRAM (OPRAM) *for a database of size N with m CPUs if Π is an $(1-\mathsf{negl}(\lambda))$-oblivious implementation of $\mathcal{F}^{N,m}_{\mathsf{PRAM}}$.*

Malicious security. The main difference between honest-but-curious and malicious security is that an adversary $\mathcal{A}$ can additionally corrupt the responses sent back to the compiler $\Pi = \{\mathcal{C}_i\}_i$. To capture this, we follow the malicious security definition of Mathialagan and Vafa [MV23] and now allow the distinguishing environment $\mathcal{A}$ to additionally control the responses back to the server. Essentially, rather than Π outputting the full list Addrs as in Experiments 8.2 and 8.3, it instead outputs adaptive PRAM queries $\widehat{\mathsf{query}}$ to the memory, which the environment $\mathcal{A}$ responds to with (possibly corrupted) $\widehat{\mathsf{data}}^*$. We formalize this as a game between the two worlds, Experiments 8.6 and 8.7 in Definition 8.8.

Experiment 8.6 $\textsc{Real}^{\mathsf{Mal}}(\Pi, \mathcal{A})$.

```
$(\widehat{\mathsf{cmd}}, \widehat{x}) \leftarrow \mathcal{A}(1^\lambda)$
while $\widehat{\mathsf{cmd}} \neq \bot$ do
    $\boxed{\widehat{\mathsf{out}} \leftarrow \bot}$
    $\widehat{\mathsf{data}}^* \leftarrow \bot$
    while $\boxed{\widehat{\mathsf{out}} = \bot}$ do
        $\boxed{(\widehat{\mathsf{query}}, \mathsf{flag}, \widehat{\mathsf{out}}) \leftarrow \Pi\left(1^\lambda, \widehat{\mathsf{cmd}}, \widehat{x}, \widehat{\mathsf{data}}^*\right)}$
        if $\mathsf{flag} = \mathsf{true}$ then return $b \leftarrow \mathcal{A}(1^\lambda)$
        $\widehat{\mathsf{data}}^* \leftarrow \mathcal{A}(1^\lambda, \widehat{\mathsf{query}})$
    end while
    $(\widehat{\mathsf{cmd}}, \widehat{x}) \leftarrow \mathcal{A}\left(1^\lambda, \boxed{\widehat{\mathsf{out}}}\right)$
end while
return $b \leftarrow \mathcal{A}(1^\lambda)$
```

Experiment 8.7 $\textsc{Ideal}^{\mathsf{Mal}}(\mathcal{F}, \mathcal{S}, \mathcal{A})$.

```
$(\widehat{\mathsf{cmd}}, \widehat{x}) \leftarrow \mathcal{A}(1^\lambda)$
while $\widehat{\mathsf{cmd}} \neq \bot$ do
    $\boxed{\mathsf{done} \leftarrow \mathsf{false}}$
    $\widehat{\mathsf{data}}^* \leftarrow \bot$
    while $\boxed{\mathsf{done} = \mathsf{false}}$ do
        $\boxed{(\widehat{\mathsf{query}}, \mathsf{flag}, \mathsf{done}) \leftarrow \mathcal{S}\left(1^\lambda, \widehat{\mathsf{cmd}}, \widehat{\mathsf{data}}^*\right)}$
        if $\mathsf{flag} = \mathsf{true}$ then return $b \leftarrow \mathcal{A}(1^\lambda)$
        $\widehat{\mathsf{data}}^* \leftarrow \mathcal{A}(1^\lambda, \widehat{\mathsf{query}})$
    end while
    $(\widehat{\mathsf{cmd}}, \widehat{x}) \leftarrow \mathcal{A}\left(1^\lambda, \boxed{\mathcal{F}(\widehat{\mathsf{cmd}}, \widehat{x})}\right)$
end while
return $b \leftarrow \mathcal{A}(1^\lambda)$
```

Definition 8.8. *For a reactive functionality $\mathcal{F}$, we say a (stateful) PRAM machine $\Pi_{\mathcal{F}} = \{\mathcal{C}_i\}_{i \in [m]}$ is a $(1-\delta)$*-maliciously secure oblivious implementation *of a reactive functionality $\mathcal{F}$ if the following two conditions hold:*

1. **Obliviousness & Correctness**: *There is a (stateful) PPT simulator $\mathcal{S}$ such that for all (stateful) PPT $\mathcal{A}$, the adversary $\mathcal{A}$ distinguishes between $\textsc{Real}^{\mathsf{Mal}}(\Pi_{\mathcal{F}}, \mathcal{A})$ (Experiment 8.6) and $\textsc{Ideal}^{\mathsf{Mal}}(\mathcal{F}, \mathcal{S}, \mathcal{A})$ (Experiment 8.7) with advantage at most δ.*
2. **Completeness**: *For all (stateful) honest-but-curious $\mathcal{A}$, with probability $1-\delta$, the compiler $\Pi_{\mathcal{F}}$ never aborts, i.e., never sets flag to true throughout the whole execution of the real experiment.*

Remark 8.9. *When combined with symmetric key encryption in a straightforward way, any $(1-\delta)$-oblivious implementation Π as in Definition 8.4 also gives a $(1-\delta-\mathsf{negl}(\lambda))$-implementation secure against honest-but-curious adversaries $\mathcal{A}$ as in Definition 8.8.*

Definition 8.10 (Maliciously secure OPRAM)**.** *We say that Π is a* $(1-\mathsf{negl}(\lambda))$-maliciously secure oblivious PRAM (OPRAM) *for a database of size N with m CPUs if Π is a $(1-\delta)$-maliciously secure oblivious implementation of $\mathcal{F}^{N,m}_{\mathsf{PRAM}}$.*

8.2 Maliciously secure OPRAM with $O(\log^2 N)$ simulation overhead

In this section, we show that combining an online PRAM memory checker with honest-but-curious OPRAM implementation gives us a maliciously secure OPRAM.

Theorem 8.11. *Suppose $\Pi = \{\mathcal{C}_i\}_{i\in[m]}$ is an $(1-\delta)$-honest-but-curious oblivious PRAM implementation. Let $\{\mathcal{M}_i\}_{i\in[m]}$ be a family of online memory checkers. Consider the family $\Pi' = \{\mathcal{C}'_i\}_{i\in[m]}$ obtained by taking $\mathcal{C}'_i$ to be the natural composition of $\mathcal{M}_i$ with $\mathcal{C}_i$. The family Π' is a $(1-\delta-\mathsf{negl}(\lambda))$ maliciously secure oblivious PRAM.*

Proof (sketch). The proof follows closely to the proof of Theorem 4.5 of Mathialagan and Vafa [MV23]. At a high level, the soundness of $\{\mathcal{M}_i\}_i$ guarantees that no incorrect response is sent back to the $\{\mathcal{C}_i\}_i$. Therefore, we can invoke the honest-but-curious obliviousness of $\{\mathcal{C}_i\}_i$ to argue that the resulting composition remains oblivious. □

Now, we recall the following theorem of Asharov et al. [AKL+22].

Theorem 8.12. *Assume the existence of one-way functions and let $N \in \mathsf{poly}(\lambda)$ and $m \leq N$. There exists a $(1-\mathsf{negl}(\lambda))$-honest-but-curious OPRAM protocol that serves a batch of m concurrrent requests with $O(m\cdot\log N)$ amortized total work, $O(\log N)$ worst-case depth, and $O(1)$ space overhead.*

Therefore, by choosing Π to be the optimal OPRAM construction of [AKL+22] (as in Theorem 1.4) and $\{\mathcal{M}_i\}_i$ to be our construction from Section 5, we obtain the following result.

Theorem 8.13. *Assume the existence of one-way functions and let $N \in \mathsf{poly}(\lambda)$ and $m \leq N$. There exists a $(1-\mathsf{negl}(\lambda))$-maliciously secure arbitrary CRCW OPRAM scheme that serves a batch of m concurrent requests with $O(m \cdot \log^2 N)$ amortized total work, $O(\log^2 N)$ worst-case depth, and $O(1)$ space overhead.*

Acknowledgements

I would like to thank Vinod Vaikuntanathan and Virginia Vassilevska Williams for their enthusiasm and guidance, and for giving valuable feedback on this manuscript. I would like to thank Neekon Vafa and Wei Kai Lin for helpful discussions, and Rahul Ilango and Yael Kirkpatrick for giving feedback on the manuscript. I would like to thank Mohsen Ghaffari and Christoph Grunau for helpful discussions on PRAMs, and Nancy Lynch for helpful discussions about distributed systems and byzantine agreement. I would also like to thank the anonymous reviewers for their detailed comments on the manuscript.

References

[AEK+17] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 251–266, 2017. 5

[Ajt02] Miklós Ajtai. The invasiveness of off-line memory checking. In *34th ACM STOC*, pages 504–513. ACM Press, May 2002. 1

[AKL$^+$22] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Optimal oblivious parallel ram. In *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2459–2521. SIAM, 2022. 6, 26, 29

[AKLS21] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. Oblivious RAM with worst-case logarithmic overhead. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 610–640, Virtual Event, August 2021. Springer, Heidelberg. 6

[BCP16] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 175–204. Springer, Heidelberg, January 2016. 1, 6, 26

[BEG$^+$91] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *32nd FOCS*, pages 90–99. IEEE Computer Society Press, October 1991. 1, 2, 3, 4, 5, 8, 13, 15, 20, 21, 22

[BNP$^+$15] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 837–849. ACM Press, October 2015. 6

[CCC$^+$16] Yu-Chi Chen, Sherman S. M. Chow, Kai-Min Chung, Russell W. F. Lai, Wei-Kai Lin, and Hong-Sheng Zhou. Cryptography for parallel RAM from indistinguishability obfuscation. In Madhu Sudan, editor, *ITCS 2016*, pages 179–190. ACM, January 2016. 1

[CCS17] T.-H. Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel RAM. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 567–597. Springer, Heidelberg, December 2017. 6

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. `https://eprint.iacr.org/2016/086`. 1

[CGLS17] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 660–690. Springer, Heidelberg, December 2017. 6

[CLT16] Binyi Chen, Huijia Lin, and Stefano Tessaro. Oblivious parallel RAM: Improved efficiency and generic constructions. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 205–234. Springer, Heidelberg, January 2016. 1

[Con22] Graeme Connell. Technology deep dive: Building a faster oram layer for enclaves. `https://signal.org/blog/building-faster-oram/`, 2022. 6

[CS17] T.-H. Hubert Chan and Elaine Shi. Circuit OPRAM: Unifying statistically and computationally secure ORAMs and OPRAMs. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part II*, volume 10678 of *LNCS*, pages 72–107. Springer, Heidelberg, November 2017. 6

[DFD$^+$21] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 655–671, 2021. 6

[DNRV09] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be? In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 503–520. Springer, Heidelberg, March 2009. 1, 4, 5

[DS83] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983. 7

[FDD12] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8, 2012. 6

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982. 7

[FLM86] Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986. 7

[FRK$^+$15] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 103–116, New York, NY, USA, 2015. Association for Computing Machinery. 6

[GHJR15] Craig Gentry, Shai Halevi, Charanjit S. Jutla, and Mariana Raykova. Private database access with HE-over-ORAM architecture. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15*, volume 9092 of *LNCS*, pages 172–191. Springer, Heidelberg, June 2015. 6

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996. 6

[Gol09] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009. 13

[LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 377–396. Springer, Heidelberg, March 2013. 6

[LO17] Steve Lu and Rafail Ostrovsky. Black-box parallel garbled RAM. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 66–92. Springer, Heidelberg, August 2017. 1

[LWN$^+$15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society Press, May 2015. 6

[Mer90] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 218–238. Springer, Heidelberg, August 1990. 1, 3, 4, 8

[MV23] Surya Mathialagan and Neekon Vafa. MacORAMa: Optimal oblivious RAM with integrity. In *To appear at CRYPTO 2023*, 2023. 1, 5, 6, 8, 11, 26, 28, 29

[NN90] Joseph Naor and Moni Naor. Small-bias probability spaces: Efficient constructions and applications. In *22nd ACM STOC*, pages 213–223. ACM Press, May 1990. 5, 20

[NR09] Moni Naor and Guy N Rothblum. The complexity of online memory checking. *Journal of the ACM (JACM)*, 56(1):1–46, 2009. 1, 4

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the Association for Computing Machinery 27*, 2, April 1980. 2005 Edsger W. Dijkstra Prize in Distributed Computing. 7

[PT11] Charalampos Papamanthou and Roberto Tamassia. Optimal and parallel online memory checking. Cryptology ePrint Archive, Report 2011/102, 2011. `https://eprint.iacr.org/2011/102`. 1, 4, 7

[RFY$^+$13] Ling Ren, Christopher W. Fletcher, Xiangyao Yu, Marten van Dijk, and Srinivas Devadas. Integrity verification for path oblivious-ram. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2013. 1, 6

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, Heidelberg, December 2011. 6

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

[WHC$^+$14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, abhi shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 191–202. ACM Press, November 2014. 6

[ZWR$^+$16] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: Efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy*, pages 218–234. IEEE Computer Society Press, May 2016. 6