# Differentially Oblivious Turing Machines

Ilan Komargodski* Elaine Shi†

July 6, 2022

**Abstract**

Oblivious RAM (ORAM) is a machinery that protects any RAM from leaking information about its secret input by observing only the access pattern. It is known that every ORAM must incur a logarithmic overhead compared to the non-oblivious RAM. In fact, even the seemingly weaker notion of differential obliviousness, which intuitively "protects" a single access by guaranteeing that the observed access pattern for every two "neighboring" logical access sequences satisfy $(\epsilon, \delta)$-differential privacy, is subject to a logarithmic lower bound.

In this work, we show that any *Turing machine* computation can be generically compiled into a differentially oblivious one with only *doubly* logarithmic overhead. More precisely, given a Turing machine that makes $N$ transitions, the compiled Turing machine makes $O(N \cdot \log \log N)$ transitions in total and the physical head movements sequence satisfies $(\epsilon, \delta)$-differential privacy (for a constant $\epsilon$ and a negligible $\delta$). We additionally show that $\Omega(\log \log N)$ overhead is necessary in a natural range of parameters (and in the balls and bins model).

As a corollary, we show that there exist natural data structures such as stack and queues (supporting online operations) on $N$ elements for which there is a differentially oblivious implementation on a Turing machine incurring amortized $O(\log \log N)$ overhead per operation, while it is known that any oblivious implementation must consume $\Omega(\log N)$ operations unconditionally even on a RAM. Therefore, we obtain the first *unconditional* separation between obliviousness and differential obliviousness in the most natural setting of parameters where $\epsilon$ is a constant and $\delta$ is negligible. Before this work, such a separation was only known in the balls and bins model. Note that the lower bound applies in the RAM model while our upper bound is in the Turing machine model, making our separation stronger.

*Hebrew University and NTT Research. Email: ilank@cs.huji.ac.li. Supported by an Alon Young Faculty Fellowship and by an ISF grant (No. 1774/20).

†Carnegie Mellon University. Email: runting@gmail.com. Supported by an NSF grant (award number CNS-1601879), an Office of Naval Research Young Investigator Program Award, and a Packard Fellowship. The work was conducted when the author was an Associate Professor in Cornell University.

# 1 Introduction

An oblivious RAM (ORAM), introduced in the seminal work of Goldreich and Ostrovsky [Gol87, Ost90, GO96], is a tool for "encrypting" the access pattern of any RAM so that it looks "unrelated" to the underlying data. It is known that any ORAM scheme must incur at least $\Omega(\log N)$ overhead, where $N$ is the size of the memory. This was first shown by Goldreich and Ostrovsky [GO96] in the balls and bins model[1] and assuming that no cryptographic assumptions are used. More recently, Larsen and Nielsen [LN18] proved the same lower bound without the two aforementioned restrictions but requiring the ORAM to support operations arriving in an online manner. In fact, a follow-up work by Jacob et al. [JLN19] showed that $\Omega(\log N)$ overhead is necessary even for obliviously implementing very specific data structures (as defined in [WNL+14]) such as stacks, queues, and more.

Apparently, logarithmic overhead is necessary even for implementing a RAM with a (seemingly) much weaker security guarantee than full obliviousness [PY19]. Persiano and Yeo [PY19] considered the notion of differentially oblivious RAM,[2] a relaxation of ORAM that only protects individual operations by guaranteeing $(\epsilon, \delta)$-differential privacy for the observed access pattern of the RAM (see Section 2.3 for a formal definition). Note that, differential privacy concerns the observed output of some algorithm. In our context, the output of an algorithm consists of the transcript of the computation: the physical memory accesses performed during the computation. Differential obliviousness was also studied in the context of *specific* functionalities by Chan et al. [CCMS19] and Beimel et al. [BNZ19]. It is shown that there are tasks for which obtaining differential obliviousness might be easier than full obliviousness. For instance, Chan et al. [CCMS19] show that there is a differentially oblivious algorithm for sorting $N$ records according to a 1-bit key while maintaining the relative ordering of records with identical keys in time $O(N \cdot \log\log N)$,[3] while [LSX19] showed a conditional $\Omega(N \cdot \log N)$ lower bound for full fledged obliviousness *in the balls and bins model*. This leaves the following natural question open.

*Is there an* unconditional *separation between obliviousness and differential obliviousness?*

Let us remark that in the above question we are interested in the most standard models and range of parameters. For RAMs, we consider the standard word-RAM where each memory word is large enough to store its own logical address, where word-level addition and Boolean operations can be done in unit cost, and where the CPU has constant number of private registers. For $(\epsilon, \delta)$-differential obliviousness, we want schemes that are secure for $\epsilon$ being a fixed constant and $\delta$ being a negligible function. For Turing machines, we allow an arbitrary number (which is fixed as part of the machine's description) of one-dimensional bi-directional infinite work tapes, where in every step the head can moved left, right, or stay in place.

## 1.1 Our Results

**Separating obliviousness from differential obliviousness.** We present a large class of functionalities that can be made differentially oblivious with only $O(\log\log N)$ overhead. The class

[1] This model assumes that each memory word is indivisible and restricts the ORAM to only move blocks around and not apply any non-trivial encoding of the underlying secret data; see Boyle and Naor [BN16].

[2] Persiano and Yeo [PY19] called this notion differentially private RAM, but we prefer to use differentially oblivious RAM to (1) relate to the notion of oblivious RAM and stress that the goal is to preserve the physical access pattern's privacy and (2) be aligned with previous work on the topic (Chan et al. [CCMS19]).

[3] Maintaining the relative ordering of records is called *stability*. Without stability, sorting records according to 1-bit keys is known to be doable (deterministically and obliviously) in linear time [AKL+20a, AKL+20b].

includes many natural and useful algorithms and data structures such as stacks and queues and therefore implies an *unconditional* separation between obliviousness and differential obliviousness.

**Theorem 1.1** (A separation; informal)**.** *For any $\epsilon, \delta > 0$, there exists a data structure (e.g., a stack or a queue) supporting $N$ operations for which:*

1. *Any oblivious implementation (even on a RAM) requires $\Omega(N \cdot \log N)$ operations;*
2. *There is an $(\epsilon, \delta)$-differentially oblivious two-tape Turing machine (defined below) that requires*

$$O\left(N \cdot (\log(1/\epsilon) + \log\log N + \log\log(1/\delta))\right)$$

*operations.*

*In particular, letting $\epsilon > 0$ be a constant and $\delta = 2^{-\log^2 N}$ (which is negligible), the number of operations incurred by the differentially oblivious machine is $O(N \cdot \log\log N)$.*

**Differentially oblivious Turing machines.** The above theorem follows from a much more general result about differentially oblivious *Turing machines*. *Oblivious* Turing machines were first introduced in 1979 by Pippenger and Fischer [PF79]. In this model, "memory accesses" correspond to the head's movements throughout the execution of the algorithm (i.e., Left, Right, or Stay). Pippenger and Fischer showed how any multi-tape Turing machine can be *obliviously* simulated by a two-tape Turing machine with a *logarithmic* slowdown in running time. More precisely, any Turing machine that makes $N$ steps can be simulated obliviously while consuming $O(N \cdot \log N)$ steps. The simulation is deterministic and perfectly oblivious: the same sequence of head movements is observed for any two inputs.

Adapting the notion of differential obliviousness to the Turing machine model, we show that any Turing machine that makes $N$ steps can be simulated by a differentially oblivious machine while making only $O(N \cdot \log\log N)$ steps. Here, neighboring sequences of head movements are ones where only one transition is different. For instance, the logical sequences of transitions {Left, Right, Left, Right} and {Left, Right, Right, Right} are neighboring.

**Theorem 1.2.** *[A differentially oblivious Turing machine; see Theorem 5.1] For any $\epsilon, \delta > 0$, any $k$-tape Turing machine that makes at most $N$ steps can be simulated by an $(\epsilon, \delta)$-differentially oblivious machine with $\max\{2, k\}$ tapes making $O(N \cdot (\log(1/\epsilon) + \log\log N + \log\log(1/\delta)))$ steps.*

As above, letting $\epsilon > 0$ be a constant and $\delta = \delta(N)$ be a particular negligible function, the number of steps incurred by the differentially oblivious machine is $O(N \cdot \log\log N)$. We note that the constant hidden in the $O$ notation depends only on the description size of the given Turing machine (i.e., its alphabet size, number of tapes, etc). Let us remark that the number of tapes we use is essentially optimal since even without any security requirements simulating a $k$-tape Turing machine for $k \geq 3$ on a $(k-1)$-tape one is not known to be possible with better than logarithmic overhead in steps (Hennie and Stearns [HS66]). Also, simulating a 2-tape machine on a single tape machine has *polynomial* overhead (Hartmanis and Stearns [HS65] for the upper bound and Hennie [Hen65] for a lower bound).

**Theorem 1.1 using Theorem 1.2.** Consider (for instance) the stack data structure on $N$ elements, supporting ("online") PUSH and Pop operations. By Theorem 1.2 and using the fact that a stack can be implemented in linear time on a Turing machine, there is an $(\epsilon, \delta)$-differentially oblivious Turing machine implementing it whose overhead is $O(\log\log N)$ for a constant $\epsilon$ and negligible $\delta$, as above. As mentioned, the logarithmic lower bound follows from Jacob et al. [JLN19].

**A lower bound.** Lastly, we observe that a lower bound of Chan et al. [CCMS19] can be tweaked to show that our construction is essentially optimal by showing that $\Omega(\log \log N)$ overhead is necessary for differential obliviousness in a natural range of parameters and in the balls and bins model.

**Theorem 1.3** (A lower bound; see Theorem 6.1)**.** *There exists an algorithmic task for which there is a Turing machine that on input of size $N$ completes it in $O(N)$ steps. On the other hand, for any $0 < s \leq \sqrt{N}$, $\epsilon > 0$, $0 < \beta < 1$, and $0 \leq \delta \leq \beta \cdot (\epsilon/s) \cdot e^{-2\epsilon \cdot s}$, any $(\epsilon, \delta)$-differentially oblivious implementation in the balls and bins model (even on a RAM) for this task must consume $\Omega(N \cdot \log s)$ steps with probability $1 - \beta$.*

In particular, for a constant $\epsilon > 0$ and $s \geq \log^2 N$, we can set $\delta = 2^{-\Omega(\log^2 N)}$ (which is negligible) and get that $\Omega(\log s) = \Omega(\log \log N)$ overhead is necessary. Note that if we want $\delta = 2^{-N^{0.1}}$, then the lower bound above says that the best we can hope for is $\Omega(\log N)$ overhead. As mentioned, with logarithmic overhead we can actually get perfect obliviousness for any Turing machine [PF79].

## 1.2 Related Work

Goldreich and Ostrovsky [Gol87, GO96] showed that any RAM that uses a memory of size $N$ and makes $T$ accesses, can be made oblivious using only $O(T \cdot \mathsf{poly} \log N)$ accesses. The resulting RAM is probabilistic and obliviousness holds against polynomial-time distinguishers assuming the existence of one-way functions. The concept of oblivious RAM has inspired an immense amount of research. One line of work, focuses on applications of such compilers to cryptography and security, including applications in cloud computing, secure processor design, multi-party computation, and more (for example, [OS97, SS13, SSS12, BNP+15, FDD12, RYF+13, MLS+13, FRK+15, WHC+14, GHJR15, LWN+15, ZWR+16, LO13, WST12]). Another line of work, focuses on improving the overhead of the compiler [SCSL11, KLO12, GM11, CGLS17, SvDS+13, WCS15]. Only recently, a couple of works [PPRY18, AKL+20a] have resolved the problem by presenting a compiler whose overhead is $O(\log N)$ (while still relying on one-way functions).

Patel et al. [PPY19] considered the natural question of what kind of security can one hope for while limiting the overhead of a RAM simulation to constant. They show a construction of an $(\epsilon, 0)$-differentially oblivious RAM with $O(1)$ overhead for $\epsilon = O(\log N)$ and also assuming that the client can store $\omega(\log N)$ records. They also proved a lower bound which quantitatively improves upon the one of [PY19] in the dependence on $\epsilon$ but is qualitatively worse since it is in the balls and bins model. Throughout this work, we focus on the setting where $\epsilon$ is a fixed constant and also that the client's storage is a constant number of blocks.

The work of Pippenger and Fischer [PF79] came in a long line of works trying to pin down the exact relation between various different computational models. One notable work is that of Hennie and Stearns [HS66] who showed that any multi-tape Turing machine can be simulated by a two-tape machine with *logarithmic* overhead. Pippenger and Fischer's result can be viewed as a similar compiler except that their resulting machine is also oblivious. Note that the result of [HS66] is the reason why one should not hope to improve the number of tapes in the resulting machine in Theorem 1.2 to two (as this task, even without privacy, is not known to be possible with less than logarithmic overhead). Simulating a 2-tape Turing machine on a single tape machine requires polynomial overhead due to Hartmanis and Stearns [HS65] and Hennie [Hen65].

Some of our ideas in the differentially oblivious Turing machine construction are reminiscent of the aforementioned differentially oblivious algorithm (in the RAM model) for *stable* tight compaction due to Chan et al. [CCMS19]. Technically, their algorithm uses similar tools from the differential privacy literature (namely, differentially private prefix sums due to Chan et al. and

Dwork et al. [CSS10, CSS11, DNPR10]) but the way they use it differ in nature from our approach. Partly, this is because our target machine is a Turing machine rather than a RAM, and therefore, standard building blocks such as oblivious sorting (which they use) are inapplicable. Second, even if we allow compiling a Turing machine to a differentially oblivious RAM (rather than insisting on Turing machine as the target machine), we still cannot directly use their techniques for constructing stable tight compaction because their techniques which rely on oblivious sorting are *offline* in nature; and thus not compatible with the *online* nature of our differentially oblivious simulation.

### 1.3 Technical Roadmap

In this overview we will focus on simulating a Turing machine with a single tape for $N$ steps. There are many complications and technical difficulties that arise in the multi-tape case, but we refer to the technical sections for details.

Given a Turing machine our goal is to hide the location of the head during the execution of the machine, in a differentially private manner. We first view this as an independent problem: given an execution of a Turing machines, output the locations of the head at pre-defined points in time throughout the execution in a differentially private manner. This is of course a necessary sub-problem to solve (as any differentially oblivious Turing machine must solve it). However, for now it is not at all obvious why it would be relevant for us, but we will explain how this algorithm helps later.

To this end, we first develop an efficiently *differentially private* algorithm for estimating the head's location at pre-defined points in time. As a first (naive) attempt, we could add a fresh Laplacian noise every time we need an estimate, but then either the privacy will suffer from a loss that depends on the number of required estimates or the efficiency of the scheme will scale with this number (since the noise would need to be really large).

To get around this, inspired by the work of Chan et al. [CCMS19], we use a differentially private prefix sum algorithm [CSS10, CSS11, DNPR10] to account for the location of the head. Recall that in the prefix sum algorithm, a stream of number arrives in an online manner and the algorithm outputs the sum of all number seen so far, after seeing every number. We set up the numbers to correspond to head movements ("Left" for -1 and "Right" for 1) and show that this approach incurs only $\mathsf{poly}\log N$ loss in privacy budget, which is good enough in terms of privacy. One challenge that we run into is that we need to implement the differentially private prefix sum algorithm on a Turing machine. It turns out that every time we need to get an estimate of the head's location (i.e., get a prefix sum), we need to pay some non-trivial factor in running time and so we need to minimize the number of such estimations. Therefore, we design our algorithm to work with only one estimate of the head's location every $\mathsf{poly}\log N$ steps and amortize the cost of this estimation while processing the next $\mathsf{poly}\log N$ steps of the Turing machine.

**Using the estimate.** Once we have a good-enough estimate of the head's location every $\mathsf{poly}\log N$ steps, all that is left is to copy the nearby positions to a smaller *oblivious* Turing machine which we use to simulate the next $\mathsf{poly}\log N$ steps. We set up the parameters in such a way that we copy enough positions around the estimated head's location to actually include the *real* head position along with the relevant tape around it to perform the next $\mathsf{poly}\log N$ steps so the above is well defined. The oblivious Turing machine that we need must provide an "initialization" procedure that allow us to start an oblivious Turing machine from an existing memory, and a "destruction" procedure which allows us to extract the memory to its original structure in the end of the execution. Pippenger and Fischer's [PF79] construction does not provide such procedures so we describe a variant that does. As an independent contribution, our new oblivious Turing Machine is described

in a language that more closely resembles the hierarchical oblivious RAM construction of Goldreich and Ostrovsky [Gol87, GO96] so it might be easier to understand for those who are familiar with the latter.

Lastly, let us remark that the above description was very high level and glossed over many technical details. For example, one technicality arises because we have to use the tapes sparingly in the compiled Turing machine to get a theorem that is tight in the number of tapes. To achieve this, we develop algorithmic tricks that allow us to reuse the same tape for multiple purposes without incurring any overhead in asymptotic running time. Other technical challenges arise because, unlike earlier explorations on differentially obliviousness [CCMS19,BNZ19], our target machine is a Turing machine rather than a RAM. This imposes additional constraints for our algorithm design since we cannot use common building blocks such as oblivious sorting. Moreover, the *online* nature of our differentially oblivious simulation also renders some previous building blocks inadequent (which we discuss more in the Related Work section). We refer to the technical sections for details.

### 1.4 Future Directions

This paper shows that a wide class of programs can be compiled into differentially oblivious counterparts with much smaller than logarithmic overhead. The class is Turing machine computations where the accessed locations are adjacent between every two memory accesses. There are several open problems that our work puts forward:

- Our upper bound has a multiplicative doubly-logarithmic term and we managed to show that it is necessary in a natural range of parameters and in the balls and bins model. It would be interesting to either show tightness of our upper bound for all settings of parameters or lift the balls and bins model restriction.
- Are there other meaningful relaxations of obliviousness that suffice for (some) applications and additionally could result with more efficient constructions?
- We focused on a statistical setting (as usually done in the context of differential privacy). Can cryptography help?

## 2 Preliminaries

For an integer $n \in \mathbb{N}$ we denote by $[n]$ the set $\{1, \ldots, n\}$. A function $\mathsf{negl}\colon \mathbb{N} \to \mathbb{R}^+$ is *negligible* if for every constant $c > 0$ there exists an integer $N_c$ such that $\mathsf{negl}(\lambda) < \lambda^{-c}$ for all $\lambda > N_c$.

### 2.1 Turing Machines

We follow the presentation of Arora and Barak [AB09] for the definition of a $k$-tape Turing machine. A tape is an infinite bi-directional line of cells, each of which can hold a symbol from a finite set called the alphabet. Each tape is associated to a tape head that can potentially read or write symbols to the tape one cell at a time. The machine's computation is divided into discrete time steps, and the head can either stay in place or move left or right one cell in each step. More formally, a Turing machine $M$ is described by a tuple $(\Gamma, Q, \Delta)$, where $\Gamma$ is a set of symbols that $M$'s tapes can contain, $Q$ is the set of $M$'s possible states, and $\Delta\colon Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, S, R\}^k$ is $M$'s transition function.

If the machine is in state $q \in Q$ and $(\sigma_1, \sigma_2, \ldots, \sigma_k)$ are the symbols currently being read in the $k$ tapes, and $\Delta(q, (\sigma_1, \ldots, \sigma_k)) = (q', (\sigma'_1, \ldots, \sigma'_k), z)$, where $z \in \{L, S, R\}^k$, then at the next

step the $\sigma$ symbols in the $k$ tapes will be replaced by the $\sigma'$ symbols, the machine will be in state $q'$, and the $k$ heads will move Left, Right, Stay in place, as given by $z$. There are additionally a read-only tape for the input and a write-only tape for the output, and perhaps a randomness tape if needed, but we ignore those when counting the number of tapes and only account for the work tapes. The space complexity of a Turing machine is the total number of cells it utilizes throughout its execution (over all tapes).

The definition above is quite robust to the choices one makes regarding the alphabet size, the number of tapes, etc, since they are all equivalent in terms of complexity up to small factors. We recall the known facts which can be found, for example, in Arora and Barak [AB09].

**Fact 2.1.** *It holds that:*

1. *Every function $f$ that is computable in time $T$ using alphabet $\Gamma$, can be computed in time $O(\log|\Gamma| \cdot T)$ using an alphabet of size $O(1)$.*
2. *Every function $f$ that is computable in time $T$ using $k$ tapes, can be computed in time $O(k \cdot T^2)$ on a single tape machine and in time $O(k \cdot T \cdot \log T)$ on a two-tape machine.*
3. *Every function $f$ that is computable in time $T$ using $k$ bi-directional tapes, can be computed in time $O(T)$ using $k$ standard (uni-directional) tapes.*
4. *Every function $f$ that is computable in time $T$ using $k$ tapes, can be computed in time $k \cdot T$ using $k$ tapes such that in each step only one of the tapes moves.*

We mention the dependence on $k$ in the above terms for explicitness even though it is a constant (and throughout the paper we treat it as constant).

In this work, we care about logarithmic factors so, by default, our Turing machine model is that of a two-tape machine. By the above, it does not matter if we consider uni-directional or bi-directional tapes. Constant factors in the alphabet size do not matter as well. All of the above only affect the constants which are hidden inside the $O$ notation.

## 2.2 Differential Privacy

Differential privacy, introduced by Dwork et al. [DMNS06], is a property of algorithms that, very roughly, guarantees "security" for a single record in the input. Namely, if the algorithm acts on the information of a set of individuals, from the output it is hard to decide whether a particular individual's information was used in the computation. This is formalized as follows. Let $A$ be a probabilistic algorithm that takes as input a dataset. Let $\mathsf{Im}(A)$ be the set of all possible outputs of $A$. The algorithm $A$ is said to be $(\epsilon, \delta)$-differentially private if for all datasets $D_0$ and $D_1$, that differ only on one entry, and all possible subsets $S \subseteq \mathsf{Im}(A)$, it holds that

$$\Pr[A(D_0) \in S] \leq e^{\epsilon} \cdot \Pr[A(D_1) \in S] + \delta,$$

where $e$ is the base of the natural logarithm.

We emphaisze that differential privacy (as defined above) provides a stataistical property which holds for all "distinguishers", even computationally unbounded ones. There are relaxations of the differential privacy guaranttee to hold for efficient (computationally bounded) distinguishers. For example, Mironov et al. [MPRV09] defined such a relaxation and used it to obtain more accurate differentially-private protocols. In this work, however, we consider only the classical statistical notion.

We refer to Dwork and Roth [DR14] for more information on differential privacy.

## 2.3 (Differentially) Oblivious Turing Machines

**Obliviousness.** Obliviousness is nowadays usually defined for RAMs and it guarantees that the access pattern of the RAM is "independent" of the underlying input. More specifically, given a RAM $M$ and an input $\mathbf{I}$, we consider a random variable $\mathsf{Accesses}(M, \mathbf{I})$ that corresponds to the ordered sequence of memory locations $M$ accesses during an execution on input $\mathbf{I}$. We then require that the distribution of $\mathsf{Accesses}(M, \mathbf{I}_1)$ is indistinguishable from $\mathsf{Accesses}(M, \mathbf{I}_2)$ for any $\mathbf{I}_1$ and $\mathbf{I}_2$ of the same length. The precise notion of indistinguishability can be either computational, statistical, or perfect, depending on the context.

A Turing machine can be thought of as a restricted version of RAM where random accesses are not allowed but any two consecutive accessed addresses must be to adjacent locations. In the context of Turing machines, this corresponds to allowing the head to move only one step right or left at a time. This directly induces an adaptation of the notion of obliviousness to the Turing machine model: the tape's head movements during the execution of the algorithm should not not leak information about the inputs. Again one can define various notions of obliviousness for this restricted model, including computational, statistical, or perfect. We consider the strong notion of *deterministic* perfect obliviousness.

**Definition 2.2** (Oblivious Turing machine)**.** *A Turing machine $M$ is said to be oblivious if for every input $x \in \{0,1\}^*$ and $i \in [N]$, the location of each of $M$'s heads at the $i$th step of execution on input $x$ is only a function of $|x|$ and $i$.*

**Differential obliviousness.** Differential obliviousness was introduced by Chan et al. [CCMS19] as a relaxation of obliviousness for RAMs. Recall that differential privacy is a framework for protecting individial records in a (large) database processed via some algorithm. Formally (as we explain in Section 2.2), this is formalized by saying that some observable event happens (almost) as likely in a database that has a particular record and in one that does not. What are those records and databases in our context?

There are two possibilities that come to mind. The first is the input for the algorithm that we execute. The second is a sequence of memory accesses / head movements. We follow previous work (e.g., [CCMS19, PY19]) and consider the latter option, adapted to the Turing machine setting.

Note that, in some cases, neighboring inputs translate to neighboring head movements, and in some cases they do not. For example, consider a streaming setting where events come in one by one, and depending on the type of the event, you either pop or push a stack. In such a scenario, neighboring inputs translate to neighboring head movements. However, in other settings, this is not necessarily the case. Imagine that the first bit of the input tells you a direction and the rest of the input is interpreted as a number. The program makes this number of steps in the specified direction. Clearly, two inputs that differ only in their first bit result with a very different sequence of head movements.

To summarize, we formalize our notion by requiring $(\epsilon, \delta)$-differential privacy for the observed access pattern/sequence of head movements. Specifically, we say that two sequences of accesses $\mathbf{I}_0$ and $\mathbf{I}_1$ are neighboring if they are of the same length and differ in exactly one location accessed. Lastly, we mention that defining neighboring inputs w.r.t the observed sequence of head movements, as we do next, still implies a privacy guarantee for the inputs using standard group privacy theorems [DMNS06].

**Definition 2.3** (Neighboring head movements)**.** *Let $M$ be a $k$-tape Turing machine. For $\mathbf{J} \in \{0,1\}^*$, let $\mathsf{Movements}(M, \mathbf{J}) \in (\{L, R, S\}^k)^*$ be the sequence of head movements that $M$ does on input $\mathbf{J}$. Two inputs $\mathbf{J}_0, \mathbf{J}_1 \in \{0,1\}^*$ are called* neighboring *if*

1. $|\mathsf{Movements}(M, \mathbf{J}_0)| = |\mathsf{Movements}(M, \mathbf{J}_1)|$ *and*

2. $\mathsf{Movements}(M, \mathbf{J}_0)$ *and* $\mathsf{Movements}(M, \mathbf{J}_1)$ *differ in exactly one location.*

*That is, letting* $(\ell_1^{b,1}, \ldots, \ell_k^{b,1}), \ldots, (\ell_1^{b,N}, \ldots, \ell_k^{b,N}) = \mathsf{Movements}(M, \mathbf{J}_b)$ *for* $b \in \{0,1\}$*, there is exactly one pair* $(i^*, j^*) \in [N] \times [k]$ *such that* $\ell_{j^*}^{0,i^*} \neq \ell_{j^*}^{1,i^*}$ *while for all other* $(i,j) \in [N] \times [k] \setminus \{(i^*, j^*)\}$*, it holds that* $\ell_j^{0,i} = \ell_j^{1,i}$.

Given this notion of neighboring inputs, we give the definition of differential obliviousness.

**Definition 2.4** ($(\epsilon, \delta)$-differentially oblivious Turing machine)**.** *A Turing machine* $M$ *satisfies* $(\epsilon, \delta)$*-differential privacy iff for any neighboring inputs* $\mathbf{J}_0$ *and* $\mathbf{J}_1$ *and any set* $S \in (\{L, R, S\}^k)^*$ *of possible sequence of head movements, it holds that*

$$\Pr\left[\mathsf{Movements}(M, \mathbf{J}_0) \in S\right] \leq e^{\epsilon} \cdot \Pr\left[\mathsf{Movements}(M, \mathbf{J}_1)) \in S\right] + \delta.$$

We emphasize that in the above the set $S$ consists of only head movements. It does not include the contents of the cells. Also, we assume that the initial head position is fixed and known. Without loss of generality, it is also okay to allow the adversary to see the accesses (but not contents) to the input, output and randomness tapes (since we can start/end by copying to/from the main tape). Not allowing access to cells' content is justified by assuming that they are already hidden from the distinguisher in some way. This could be by some form of encryption. This means that we only care about the cost of making the access pattern/head movements (differentially) oblivious and ignore the underlying method used to hide the cells' content. This is the standard adversarial model considered in the oblivious RAM literature.

Looking forward, we achieve Definition 2.4 by giving a compiler from (deterministic) Turing machine TM to a (randomized) Turing machine TM'. The latter TM' has a security property that depends on the inputs being fed into it. Specifically, if TM' is executed on two inputs that result with neighboring head movements on *the original machine TM*, then the resulting head movements in TM' will be indistinguishable. Previous works, e.g., [PY19], have the same definition but with head movements replaced with logical memory access sequence.

# 3 Estimating Heads' Locations

In this section, we present an algorithm running on a Turing machine that outputs estimates to the location of the heads in a Turing machine computation. More precisely, the input to the algorithm is a sequence of movements of the heads of the machine (i.e., Left, Right or Stay for each tape), and it outputs an estimate to the location of the head in a-priori fixed intervals of time in an online fashion. The algorithm (1) outputs estimates which are not too far from the true position of the head at a given time, (2) the estimates are differentially private, (3) the algorithm's head movements themselves are oblivious (i.e., data-independent), and (4) the algorithm is very efficient. The interval at which we output an estimate on the location of the heads is a parameter, denoted $p$.

**Theorem 3.1.** *Fix* $k \in \mathbb{N}$*. There exists an algorithm* $\mathsf{EstimateHead}_{\epsilon,\delta}$ *such that for any* $\epsilon, \delta > 0$*, the following holds. Fix any stream* $\mathbf{a} = a_1, a_2, \ldots, a_N \in \{L, S, R\}^k$ *that corresponds to the movements of the heads of a* $k$*-tape Turing machine. Treating* $L$ *as -1,* $S$ *as 0, and* $R$ *as 1, let* $\sigma_i = \sum_{j=1}^{i \cdot p} a_i$ *for* $i \in [N/p]$ *(i.e., the true position of the heads every* $p$ *steps). Let* $\{\tilde{\sigma}_i\}_{i \in [N/p]}$ *denote a possible output of the algorithm* $\mathsf{EstimateHead}_{\epsilon,\delta}$ *when fed* $\mathbf{a}$ *as an input in an online fashion. It holds that:*

1. **Utility**: *With probability 1 over the randomness of* $\mathsf{EstimateHead}_{\epsilon,\delta}$*, it holds that*

$$\max_{i\in[N/p]} |\tilde{\sigma}_i - \sigma_i| \in O\left((1/\epsilon)\cdot \log^{1.5} N \cdot \log(1/\delta)\right).$$

2. **Differential privacy**: $\mathsf{EstimateHead}_{\epsilon,\delta}$ *is* $(\epsilon,\delta)$*-differentially private (as per Section 2.2). Here, neighboring sequences are defined in the natural way by allowing only one of the* $k$ *indices in one of the* $N$ $a_i$*'s to differ between the two sequences.*

3. **Obliviousness**: *The algorithm itself is perfectly oblivious.*

4. **Efficiency**: *The algorithm runs in time* $O\left(N + (N/p)\cdot(\log N)\right)$.

**Proof.** The algorithm builds on the differentially private prefix-sum algorithm of Chan et al. [CSS10, CSS11] and Dwork et al. [DNPR10]. Their algorithms address the problem of continuously estimating the prefix sums of elements in a given stream of numbers while maintaining differential privacy. We follow the presentation of these algorithm from Dwork and Roth [DR14, §12.3]. The algorithm is given a stream of numbers $\mathbf{b} = b_1, b_2, \ldots, b_N \in \{-1, 0, 1\}$ that the algorithm sees in an online fashion. The algorithm outputs, after seeing $b_1, \ldots, b_j$ an approximation of $\sum_{i=1}^{j} b_i$. This task is almost what we need to prove our theorem for $k = 1$ (i.e., the machine has one tape). Indeed, a movement left (resp. right) can be interpreted as -1 (resp. 1) and staying in place corresponds to seeing 0. The location of the head is exactly the sum of those numbers. Additionally, it is not hard to observe that their algorithm is in fact oblivious (see below). Nevertheless, the running time of their algorithm on a Turing machine is $O(N \cdot \log N)$ (see below). So, we need to (1) extend the algorithm to handle any $k \geq 1$ tapes and (2) show how to implement it in the specified running time on a Turing machine. Since both goals are somewhat non-trivial to achieve, let us first briefly recall their algorithm and state its guarantees, and then describe our modifications.

Assume that $N$ is a power of 2 (for simplicity and without loss of generality). We associate the $N$ numbers to leaves of a full binary tree and then label each node in the tree with an "interval". The $i$th leaf (for $i \in [N]$) is labeled with $[i, i]$. An internal node is labeled with the interval that is the union of the intervals associated with its children. Now, with each node, labeled $[s, t]$ in this tree, we associate a noisy count that approximates the sum of the values seen in positions $s, s+1, \ldots, t$ by adding noise from the appropriate distribution. In [CSS10, CSS11, DNPR10] the added noise was sampled from $\mathsf{Lap}((1+\log_2 N)/\epsilon)$, where $\mathsf{Lap}(s)$ denotes the (continuous) Laplace distribution with mean 0 and variance $2s^2$. To output $\tilde{\sigma}_i$ (i.e., the approximation of $\sum_{j=1}^{i\cdot p} a_j$), we write $t = i \cdot p$ in binary to find at most $\log_2 N$ intervals whose union is $[1, t]$, and compute the sum of the corresponding noisy counts. These intervals are associated to the nodes which are called *the frontier*. This algorithm satisfies $(\epsilon, 0)$-differential privacy and satisfies the following utility property where $\delta$ is a parameter: With probability $1 - \delta$ over the randomness of the algorithm,

$$\max_{i\in[N/p]} |\tilde{\sigma}_i - \sigma_i| \leq O\left((1/\epsilon)\cdot \log^{1.5} N \cdot \log(1/\delta)\right).$$

It is easy to turn the utility property to be satisfied with probability 1 by outputting the exact prefix sum in the clear whenever the error in the output is too large. This causes the algorithm to be $(\epsilon, \delta/2)$-differentially private, as needed. From the description of the protocol, the only step that depend on the input sequence is the one where we compute the sum of the noisy counts along the $\log_2 N$ intervals; all other steps depend only on $N$ (and fresh randomness). Since computing the sum can be easily made (perfectly) oblivious, we get that the algorithm itself is (perfectly) oblivious. This property will be preserved throughout the following modifications.

**Handling multiple tapes.** We extend the algorithm to handle $k$ tapes by maintaining $k$ prefix sums computed in parallel. This clearly does not hurt utility or obliviousness and only incurs a $k$ factor in running time. However, naively, it also incurs a $k$ factor in differential privacy. Nevertheless, we observe that considering any two neighboring sequences of inputs, $k-1$ of the tapes will have the exact same access pattern while only one will differ in one position, and so this extension, in fact, does not incur a $k$ factor in differential privacy.

**Running time.** The main challenge is to maintain an updated version of the noisy counts associated to the nodes in the frontier. Recall that the frontier is of size $\log_2 N + 1$. Naively, with the above algorithm, computing the frontier at time $i+1$ from the frontier at time $i$ may cost up to $O(\log N)$ work which is too expensive for us. However, recall that we do not need a prefix sum after every $a_i$, but rather we want to output one after every $p$ inputs. So, instead of having a full binary tree where the leaves correspond to each input, we consider a full binary tree where each leaf corresponds to a sequence of $p$ inputs and it is labeled by their sum. The depth of this tree is $\log_2(N/p)$ and the point is that we need to compute the "next" frontier (which costs about $\log_2 N$) only once every $p$ operations, so the total cost is $O(k \cdot (N/p) \cdot \log N)$ plus the time it takes to aggregate the sum itself which is $O(k \cdot N)$, as needed. ■

**Remark 3.2** (Sampling from $\mathsf{Lap}$)**.** *We emphasize that the above algorithm assumes that a Turing machine is capable of sampling from* $\mathsf{Lap}(\cdot)$ *in* $O(1)$ *time. This is assumed for simplicity of presentation. However, it is possible to efficiently compute an estimate of this distribution on a standard Turing machine. The cost of this approximation is good enough to obtain (asymptotically) the same final result in Theorem 5.1. Therefore, the assumption being made in this section is without loss of generality in the context of our main result. See details in Appendix A.*

# 4 Oblivious Turing Machines

A classical result by Pippenger and Fischer [PF79] shows that any Turing machine computation can be made perfectly oblivious (i.e., Definition 2.2) on a two-tape machine with amortized logarithmic overhead. More precisely, any Turing machine that makes at most $N$ steps can be made perfectly oblivious while making $O(N \cdot \log N)$ steps.

In our application we need an oblivious Turing machine which support two additional properties. The first, called "initialization", is that one can initialize an oblivious Turing machine with a given memory (as opposed to starting off with an empty memory). The second, called "destruction", returns the state of the memory in a linear fashion.

Our construction is similar in spirit to the one of Pippenger and Fischer [PF79]. However, we present it in a language that more closely resembles the hierarchical oblivious RAM construction of Goldreich and Ostrovsky [Gol87, GO96] so it might be easier to understand for those who are familiar with the latter.

**Theorem 4.1** (Oblivious Turing machine, revisited)**.** *Any $k$-tape Turing machine that makes at most $N$ steps can be executed obliviously on a two-tape Turing machine with $O(N)$ space and with $O(N \cdot \log N)$ steps. Additionally, the machine supports initialization and destruction.*

**Proof.** We will present the main idea in the special case where the given Turing machine has only a single tape and the resulting machine will have $\ell = \lceil \log N \rceil$ tapes. Later, we will explain how to handle multiple tape machines in the input and simulate them obliviously with just two tapes (at the same cost).

We refer to each of the $\ell$ tapes as a "level" (analogously to levels in [GO96]'s hierarchical ORAM construction). Level 1 has capacity to hold $2 \cdot 3 = 6$ cells, level 2 has capacity to hold $4 \cdot 3 = 12$ cells, and in general level $i$ has capacity to hold $2^i \cdot 3$ cells. Each level is split into virtual "blocks", each holding $2^i$ cells. Every such block starts at an address $j$ such that $j \bmod 2^i = 0$. That is, level 1's blocks start at even addresses, level 2's blocks start at addresses divisable by 4, and so on. At the beginning, when a level is built, the head of that level points to the middle block; for concreteness, say it always points to the leftmost element in the block.

Recall that an access consists of a read or write operation as well as a direction to move the head to, left or right. When an access is made, all cells in the middle block of the smallest level are scanned, all of which being ignored except a single cell where the real head points to and is therefore the one we need to read from or write to. The original TM head position is stored in a designated register and it is updated upon each access.

Every level $i$ is rebuilt every $2^{i-1}$ logical steps. That is, level 1 is rebuilt after every step, level 2 is rebuilt after 2 steps, and so on. Here is how a rebuilt of level $i$ works:

1. Level $i$ writes its contents to level $i+1$ in a block-wise manner. That is, each block in level $i$ puts itself in the right place (out of 6) in block $i+1$. To do this obliviously, each block in level $i$ tries to place itself in one of the possible positions and only one of these attemps will not be dummy. (This costs $O(1)$ head movements.)

2. The address of the middle block for level $i$ is being recomputed based on the original TM head position and then level $i$ gets updated contents from level $i+1$ in a block-wise manner. That is, each block in level $i$ gets updated information from the right place (out of 6) in block $i+1$. This is also done via $O(1)$ head movements in brute-force.

Correctness follows immediately by description. Perfect obliviousness follows from the fact that the head's movements are deterministic and a-priori fixed. For efficiency, consider any sequence of $N$ steps. A read or a write are done at $O(1)$ operations cost by just accessing the middle block in the first level. It remains to account for the cost of the reorganization steps. By description and recalling that the size of level $i$ is $2^i \cdot 3$, the total amount of steps performed by the oblivious Turing machine is bounded by

$$\sum_{i=1}^{\log N} O\left(\left\lceil \frac{N}{2^i} \right\rceil \cdot 2^i\right) \in O(N \cdot \log N).$$

We now explain how to remove the simplifying assumptions we had, the first being that the input machine has only one tape and the second being that the resulting oblivious machine uses $\log N$ many tapes. Let us first handle the former, letting $k$ be the number of tapes used by the input machine. We use an encoding trick. We encode the $k$ tapes into a single tape by first placing all the first cells from each tape, then the second cell, and so on. Each "track" will have its own head marker. By the construction of the oblivious Turing machine, all the tracks can be processed simultaneously (recall that our head movement sequence is deterministic), incurring a $k$ multiplicative factor.

Now, we explain how to modify our Turing machine to use only two tapes. As a first step, let us place the different levels one after the other on the single tape. Naively, this incurs a blowup in running time due to the reorganization steps. Indeed, in the reorganization steps, we need to scan two levels "in parallel" as cyclic buffers. The only way to do this with a single tape is by moving back and forth in the tape which is too expensive. This is where we will use the second tape. When such a "parallel" scan is needed, we will copy one of the levels to the second tape, do

the "parallel" scan by scanning both tapes in parallel, and then copy it back. This only incurs a constant overhead.

**Initialization and destruction.** In our application, we will need an oblivious Turing machine with two additional features so we explain how to implement them next. The first is that we need to support initialization with a given memory which might not necessarily be empty. We implement this by starting with an empty memory, as described above, and modifying the memory one element at a time. If the number of steps that we eventually perform on the Turing machine is about the size of the initial memory, the cost of this step will be amortized away.

The second feature is a destruction procedure which outputs the memory at the end of the computation in a linear fashion. This is not so immediate since our construction does not store the memory in a linear fashion. Recall that our construction satisfies that at the end of every $2^i$ steps, the cells corresponding to the level $T_i$ store the content of the tape at positions $[p-2^{i-1} : p+2^{i-1}]$. This means that if "destruct" is invoked after a power-of-2 many steps, the memory is stored exactly in the cells corresponding to some level and one can make one linear scan to extract those elements and put them one next to the other. If destruct is invoked after some other number of steps, we need to modify this procedure slightly by collecting the most updated memory values of each cell from the appropriate level (now, the most updated values are spread amongst different cells). This again can be done by a single scan. ■

# 5 A Differentially Oblivious Turing Machine

In this section, we describe our transformation from any Turing machine into a differentially oblivious one.

**Theorem 5.1.** *For any $\epsilon, \delta > 0$, any $k$-tape Turing machine that makes at most $N$ steps and consumes $S$ space can be transformed into an $(\epsilon, \delta)$-differentially oblivious $\max\{2, k\}$-tape Turing machine that makes $O(N \cdot (\log(1/\epsilon) + \log\log N + \log\log(1/\delta)))$ steps and consumes $O(S + (1/\epsilon) \cdot \log^2 N \cdot \log(1/\delta))$ space.*

The construction of the differentially oblivious Turing machine uses the oblivious Turing machine construction from Section 4 and the head's location estimation algorithm from Theorem 3. We will present the construction in steps. We first assume that the input machine uses only one tape and the resulting machine will use many tapes. Then, we will explain how to get rid of both simplifying assumptions and therefore obtain Theorem 5.1.

## 5.1 From One Tape to Four Tapes

Assume first that the given machine, $M$, uses only a single tape. We first present a construction that compiles $M$ into a differentially oblivious Turing machine $\mathsf{do}M$ with 4 tapes. Fix $\epsilon, \delta > 0$ for the rest of this section.

**Tape allocation.** The resulting Turing machine, $\mathsf{do}M$, will consist of four tapes, numbered 1, 2, 3, and 4, in the following order:

1. One tape to simulate the input Turing machine computation (recall that we assumed that the input machine has only one tape).

2. Two tapes for running an oblivious Turing machine (according to Section 4).

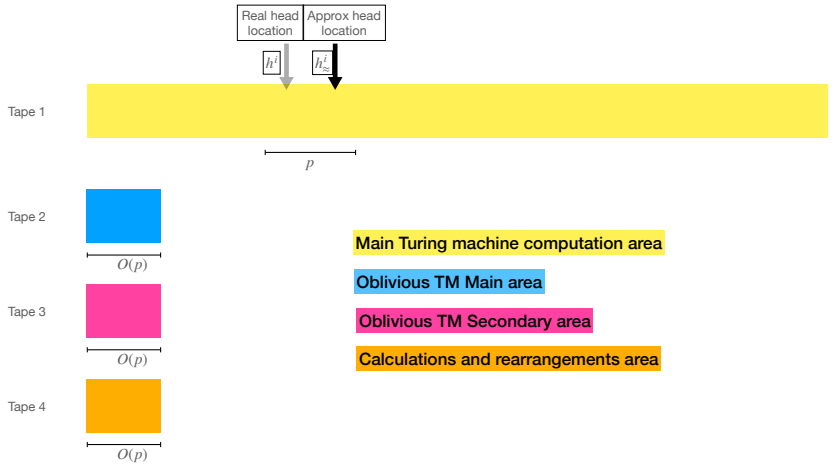


**Figure 1:** An illustration of the differentially oblivious TM tape configuration for compiling a single-tape TM into a 4-tape machine.

3. One tape to compute differentially private head's location estimation algorithm (according to Section 3).

**The algorithm.** As mentioned, we use the oblivious Turing machine implementation from Section 4 but since its overhead is logarithmic (in the running time of the non-oblivious machine), we do not want to apply it directly on our machine. Instead, we are going to break down the computation of the original machine into epochs and invoke the oblivious machine only within epochs. Concretely, we split the computation of $M$ into epochs of size

$$p \triangleq (1/\epsilon) \cdot \log^2 N \cdot \log(1/\delta).$$

Each such epoch will be executed in its own "fresh" oblivious Turing machine and so the overhead will only be a doubly logarithmic factor in $N$. Next, we explain how $\mathsf{do}M$ works.

$\underline{{}_{1\to 4}\mathsf{do}M_{\epsilon,\delta}}$:

1. Set $h^0_{\approx} = 0$ to be the initial approximate position of the head (it is equal to the real position).
2. Break the $T$-step computation into epochs of $p$ steps of computation. For epoch $i = 1, \ldots, N/p$, do:
   (a) Copy an area of size $4p + 1$ around $h^{i-1}_{\approx}$, namely $[h^{i-1}_{\approx} - 2p, h^{i-1}_{\approx} + 2p]$ to the oblivious Turing machine (Theorem 4.1). Perform the next $p$ steps of computation there. At the end of the epoch, copy the state of these $4p + 1$ cells back to the main tape.
   (b) In parallel, keep track of the movements of the head and count the offset of the head compared to the previous location, $h^{i-1}_{\approx}$. At the end of the epoch, invoke the differentially private head's location estimation algorithm (Theorem 3.1) with privacy parameters $\epsilon$ and $\delta$ to update the location of the head $h^i_{\approx}$.

See an illustration in Figure 1.

**Theorem 5.2.** *For any $\epsilon, \delta > 0$ and given any single-tape Turing machine $M$ that makes at most $N$ steps and consumes $S$ space, the 4-tape machine $\mathsf{do}M_{\epsilon,\delta}$ is $(\epsilon, \delta)$-differentially oblivious, makes at most $O(N \cdot (\log(1/\epsilon) + \log\log N + \log\log(1/\delta)))$ steps and consumes $O(S + (1/\epsilon) \cdot \log^2 N \cdot \log(1/\delta) \cdot (\log(1/\epsilon) + \log\log N + \log\log(1/\delta))$ space.*

**Proof.** We first prove correctness, ignoring obliviousness. Consider any sequence of operations. At any point in time, the oblivious Turing machine contains $2p + 1$ memory cells and performs all necessary operations within. For correctness, by description, it is enough to show that the $p$ operations are indeed contained within those $2p + 1$ cells. Indeed, for this to hold it is enough to argue that $h^i_{\approx}$ is close enough to the real location of the head: $h^i_{\approx} - 2p \leq h^i - p$ and $h^i_{\approx} + 2p \geq h^i + p$, where $h^i$ is the true location of the head. In other words, we need to show that

$$|h^i_{\approx} - h^i| \leq p.$$

By the utility property of the head's location estimation algorithm (Theorem 3.1), we know that $|h^i_{\approx} - h^i| \leq (1/\epsilon) \cdot \log^{1.5} N \cdot \log(1/\delta)$ (this is the upper bound on the additive error of each head's location estimation for every $i$). Now, the above inequality follows by recalling that $p = (1/\epsilon) \cdot \log^2 N \cdot \log(1/\delta)$.

To prove $(\epsilon, \delta)$-differential obliviousness, consider any two sequences of operations $\mathbf{I}_0$ and $\mathbf{I}_1$ that differ at one operation. Consider the random variable $\tilde{I}_b$ corresponding to the physical tape heads locations on input $\mathbf{I}_b$ for $b \in \{0, 1\}$. Say the two sequences $\mathbf{I}_0$ and $\mathbf{I}_1$ differ in the $i$th operation and are otherwise identical. Then, the first $i - 1$ operations result with identical distributions of head locations in both executions (as all the underlying building blocks are perfectly oblivious). The only difference is in the $i$th operation. There, the head's locations might differ due to a different distribution of the head's location estimation algorithm (Theorem 3.1). However, we are guaranteed that this algorithm is $(\epsilon, \delta)$-differentially oblivious. The rest of the heads' movements are perfectly oblivious: the oblivious Turing machine is perfectly oblivious, the head's location estimation algorithm itself is perfectly oblivious, and the other operations that we do in the implementation of $\mathsf{do}M$ are trivially oblivious.

Lastly, we analyze efficiency by counting the total amount of work and space required to handle any sequence $N$ operations that consume $S$ space. Step 3a costs $O(p \cdot \log p)$ operations and space due to Theorem 4.1 (the rest of the operations can be implemented in $O(p)$ time and space). Computing the differentially private head's location estimation in Step 3b takes overall $O(N + (N/p) \cdot (\log N)) < O(N)$ time due to Theorem 3.1 (i.e., constant amortized work per access). Otherwise, simulating the original computation and accounting for the location of the head, requires $O(N)$ work and space. Overall, over $N$ operations, the total space is $O(N + p \cdot \log p)$ and the work is bounded by $O(N \cdot \log p)$. Plugging in $p = (1/\epsilon) \cdot \log^2 N \cdot \log(1/\delta)$ completes the proof. ∎

## 5.2 From One Tape to Two Tapes

In this section we show how to obtain the same result as in the previous section (i.e., Theorem 5.2), except that our resulting Turing machine will only use two tapes (instead of four). One tape will be used for the simulation of the original Turing machine plus one of the tapes of the oblivious Turing machine and the other tape will be used to perform the head's location estimation algorithm and the other tape of the oblivious Turing machine.

Recall that the tapes used for the oblivious Turing machine, both consume about $p$ space, but one interacts with the main tape (call it tape $o\mathsf{TM}_1$) and the other acts as a scratch pad and the values that are written there are never accessed outside of the oblivious Turing machine implementation (call it tape $o\mathsf{TM}_2$). We will merge tape $o\mathsf{TM}_2$ into the tape that simulates the original computation, and tape $o\mathsf{TM}_1$ to the tape that computes the prefix sums.

**Tape 1 (Main Tape).** This tape will consist of the main computation tape as well as a blank area which is used for tape $o\mathsf{TM}_2$ of the oblivious Turing machine. We use the fact that the required space for the oblivious Turing machine is $O(p)$ cells and so we will maintain such a "space of blanks" which will not be too far from the real position of the head and will be used whenever a new oblivious Turing machine is instantiated. Let us denote by $S_{o\mathsf{TM}} = O(p)$ the space consumption of the oblivious Turing machine. Our first tape, the one that simulates the computation of the original Turing machine, will maintain the invariant that in distance $S_{o\mathsf{TM}}$ from the location of the approximate head $h_{\approx}$ to the right, there are $S_{o\mathsf{TM}}$ blank cells (the last blank cell has distance $2S_{o\mathsf{TM}}$ from $h_{\approx}$). If this invariant holds, then whenever an epoch begins, we can move to the blank area and use it as the oblivious Turing machine tape. At the end of the epoch, we can go back to where we were. Since we perform $p$ operations inside the oblivious Turing machine, the amortized cost of moving back and forth is $O(1)$ per operation which is what we need.

We thus need to explain how to maintain the above invariant. The idea is to move the blank area together with the location of the head once every epoch. Namely, once we update the approximate position of the head $h_{\approx}$, we will also move the blank area appropriately so that its distance from the new $h_{\approx}$ is as we require. Moving the blank area, as above, can be done simply in time $O(p)$ using a designated size $O(p)$ space in the other tape (tape 2)—this is done by moving the area that needs to go to the blank area to tape 2 (and replacing it with blanks), and then copying by moving both heads "in parallel".

**Tape 2 (Secondary Tape).** This tape will consist of three areas, each of size $O(p)$. One of these areas will be used for moving the blank area in tape 1, as we explained above. Another area is for the computation of the head's location estimation—this algorithm has state of size $O(\log N) < O(p)$ (which contains a frontier of a tree of noisy sums per interval). The third part is for tape $o\mathsf{TM}_1$ of the oblivious Turing machine (which also uses $O(p)$ space).

The first and second parts in this tape are accessed at the end of every epoch and some computation of length $O(p)$ is performed on each of them (either updating the prefix sum or moving data around). Since each epoch handles $O(p)$ operations, the amortized cost of this part is $O(1)$ per operation of the original machine. The third part, in contrast, is accessed throughout the epoch, and there we get $O(\log p)$ overhead per operation.

$\underline{1\to 2\mathsf{do}M'_{\epsilon,\delta}}$:

1. Set $h^0_{\approx} = 0$ to be the initial approximate position of the head (it is equal to the real position).
2. Allocate an empty area of $S_{o\mathsf{TM}} = O(p)$ cells in Tape 1 with some large enough hidden constant. Call this "the blank area". This area will be $S_{o\mathsf{TM}}$ away from $h^0_{\approx}$.
3. Break the $T$-step computation into epochs of $p$ steps of computation. For epoch $i = 1, \ldots, N/p$, do:
   (a) Copy an area of size $4p+1$ around $h^{i-1}_{\approx}$ from Tape 1, namely $[h^{i-1}_{\approx} - 2p, h^{i-1}_{\approx} + 2p]$, to the beginning of Tape 2. Perform the next $p$ steps of computation using an oblivious TM, treating Tape 2 as the main tape and the blank area in Tape 1 as the scratch tape. At the end of the epoch, copy the state of these $4p+1$ from Tape 2 back to the right location in Tape 1.
   (b) In parallel, keep track of the movements of the head and count the offset of the head compared to the previous location, $h^{i-1}_{\approx}$. At the end of the epoch, invoke the differentially private head's location estimation algorithm (Theorem 3.1) with privacy parameters $\epsilon$

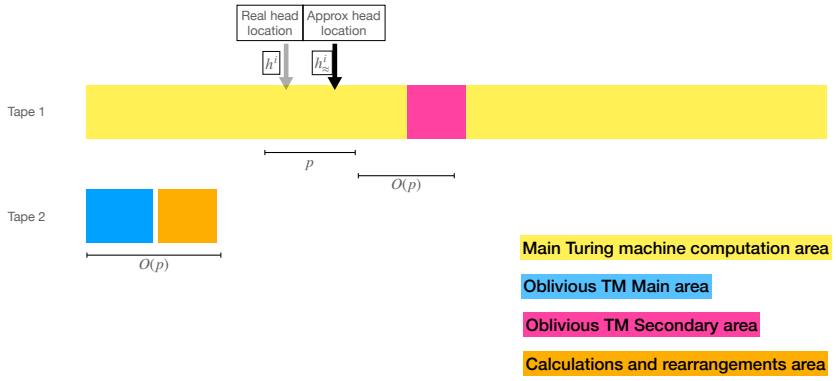


**Figure 2:** An illustration of the differentially oblivious TM tape configuration for compiling a single-tape TM into a 2-tape machine.

and $\delta$ to update the location of the head $h^i_{\approx}$. This is done in using a designated area in Tape 2 (residing after the area used to the oblivious TM computation). Lastly, move the blank area in Tape 1 so that the invariant that it resides $S_{o\mathsf{TM}}$ away from $h^0_{\approx}$ is maintained (for this, use a designated area in Tape 2).

See an illustration in Figure 2.

### 5.3 From $k$ Tapes to $k$ Tapes (for $k \geq 2$)

This extension is done by making two changes. First we use $k$ tapes to simulate the computation of the original $k$-tape machine (instead of just a single tape). Second, we use the algorithm for estimating the head's position which works for $k$ tape machines (Theorem 3.1)—this incurs an overhead of $k$ operations per step. Recall that this algorithm just runs the algorithm for estimating the head's position of a single tape $k$ times (independently). It remains to explain where we execute this algorithm and also where we execute the oblivious Turing machine since now we do not have an extra work tape.

More precisely, we modify each tape to have *two* "blank areas", each as above. The first one will act as the "Main Tape" in the above construction and the second one acts as the "Secondary Tape" for another tape. Concretely, tape $(i+1) \bmod k$ acts as the "Secondary Tape" of tape $i$ (for all $i \in [k]$). That is, the first blank area of the head of tape $i$, is used to simulate the computation of tape $i$ in the original machine and also to execute tape $o\mathsf{TM}_2$ of the oblivious Turing machine when simulating tape $i$ (this is exactly the same usage of the blank area as above). The second blank area of tape $i$ consists of three areas: (1) an area used to maintain the blank areas in tape $(i+1) \bmod k$, (2) an area used for the computation of the head's location estimation of tape $(i+1) \bmod k$, and (3) tape $o\mathsf{TM}_1$ of the oblivious Turing machine when simulating tape $(i+1) \bmod k$.

See an illustration in Figure 3.

## 6 Lower Bound

In this section we prove that our differentially oblivious Turing machine is optimal in terms of overhead in a natural range of parameters. Specifically, we prove the following theorem.

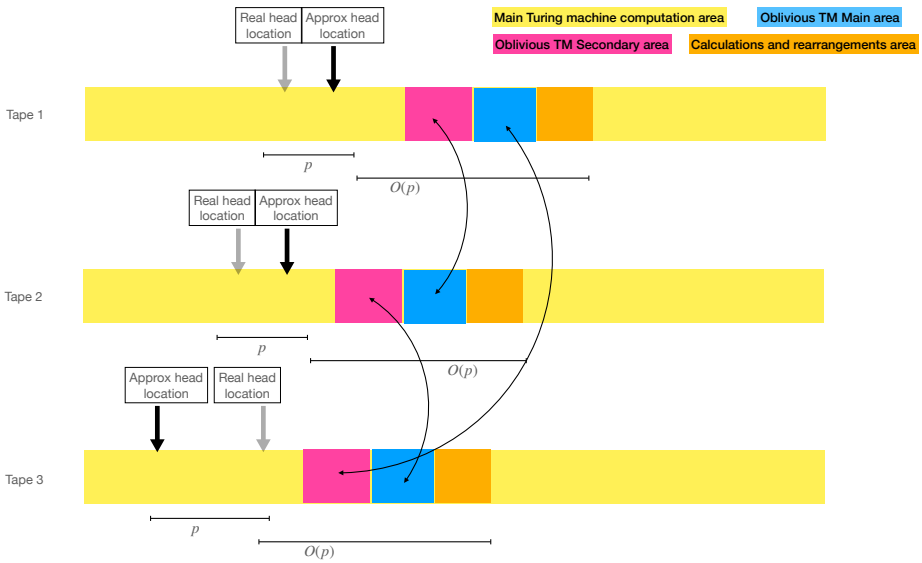


**Figure 3:** An illustration of the differentially oblivious TM tape configuration for compiling a $k$-tape TM into a $k$-tape machine.

**Theorem 6.1.** *There exists an algorithmic task for which there is a Turing machine that on input of size $N$ completes it in $O(N)$ steps. On the other hand, for any $0 < s \leq \sqrt{N}$, $\epsilon > 0$, $0 < \beta < 1$, and $0 \leq \delta \leq \beta \cdot (\epsilon/s) \cdot e^{-2\epsilon \cdot s}$, any $(\epsilon, \delta)$-differentially oblivious implementation (even on a RAM and in the balls and bins model) for this task must consume $\Omega(N \cdot \log s)$ steps with probability $1 - \beta$.*

**Proof.** In the work of Chan et al. [CCMS19] the following theorem concerning the required overhead to stably sort a set of balls according to associated 1-bit keys while maintaining differential obliviousness. Here, we assume that the balls are opaque and so no non-trivial encoding on them can be done [BN16].

**Theorem 6.2** (Theorem 4.7 in [CCMS19])**.** *Let $0 < s \leq \sqrt{N}$. Suppose $\epsilon > 0$, $0 < \beta < 1$, and $0 \leq \delta \leq \beta \cdot (\epsilon/s) \cdot e^{-2\epsilon \cdot s}$. Then, any (even randomized) stable sorting algorithm for balls according to associated 1-bit keys in the RAM model that is $(\epsilon, \delta)$-differentially oblivious must have some input, on which it incurs at least $\Omega(N \cdot \log s)$ memory accesses with probability at least $1 - \beta$.*

The task of stably sorting $N$ balls according to associated 1-bit keys can be implemented using a Turing machine in $O(N)$ steps. Consider an input of the form $(k_1, v_1), \ldots, (k_N, v_N)$, where $k_i \in \{0, 1\}$ is a 1-bit key and $v_i$ is the $i$th ball. The idea is to scan the input from the beginning and whenever we see an element $(k_i, v_i)$ we do one of the following. If $k_i = 0$, we write $(k_i, v_i)$ to the next position in the output tape. If $k_i = 1$, we write it to the next position in the work tape. After we finish scanning the input, we scan the output again and write all elements from first to last into the output tape. It is immediate that this algorithm is correct and has $O(N)$ running time. ■

## Acknowledgement

We thank Hubert Chan and Kai-Min Chung for useful discussions.

## References


[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009.

[AKL+20a] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In *Advances in Cryptology - EUROCRYPT*, pages 403–432, 2020.

[AKL+20b] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Enoch Peserico, and Elaine Shi. Oblivious parallel tight compaction. In *1st Conference on Information-Theoretic Cryptography, ITC*, pages 11:1–11:23, 2020.

[BN16] Elette Boyle and Moni Naor. Is there an oblivious RAM lower bound? In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, ITCS*, pages 357–368, 2016.

[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 *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 837–849. ACM, 2015.

[BNZ19] Amos Beimel, Kobbi Nissim, and Mohammad Zaheri. Exploring differential obliviousness. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM*, pages 65:1–65:20, 2019.

[CCMS19] T.-H. Hubert Chan, Kai-Min Chung, Bruce M. Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2448–2467. SIAM, 2019.

[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 *Advances in Cryptology - ASIACRYPT*, pages 660–690, 2017.

[CSS10] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. In *Automata, Languages and Programming, 37th International Colloquium, ICALP*, pages 405–417, 2010.

[CSS11] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. *ACM Trans. Inf. Syst. Secur.*, 14(3):26:1–26:24, 2011.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography, Third Theory of Cryptography Conference, TCC*, pages 265–284, 2006.

[DNPR10] Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N. Rothblum. Differential privacy under continual observation. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC*, pages 715–724. ACM, 2010.

[DR14] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[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. ACM, 2012.

[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*, pages 103–116. ACM, 2015.

[GHJR15] Craig Gentry, Shai Halevi, Charanjit Jutla, and Mariana Raykova. Private database access with HE-over-ORAM architecture. In *International Conference on Applied Cryptography and Network Security*, pages 172–191. Springer, 2015.

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *Automata, Languages and Programming - 38th International Colloquium, ICALP*, pages 576–587, 2011.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.

[Gol87] Oded Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, STOC*, pages 182–194, 1987.

[Hen65] F. C. Hennie. One-tape, off-line Turing machine computations. *Inf. Control.*, 8(6):553–578, 1965.

[HS65] Juris Hartmanis and Richard E Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[HS66] F. C. Hennie and Richard Edwin Stearns. Two-tape simulation of multitape turing machines. *J. ACM*, 13(4):533–546, 1966.

[JLN19] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2439–2447. SIAM, 2019.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 143–156, 2012.

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

[LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *Theory of Cryptography*, pages 377–396. Springer, 2013.

[LSX19] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can we overcome the n log n barrier for oblivious sorting? In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 2419–2438, 2019.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *IEEE Symposium on Security and Privacy*, 2015.

[MLS+13] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. PHANTOM: practical oblivious computation in a secure processor. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 311–324, 2013.

[MPRV09] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil P. Vadhan. Computational differential privacy. In *Advances in Cryptology - CRYPTO*, pages 126–142, 2009.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, STOC*, pages 294–303, 1997.

[Ost90] Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 514–523, 1990.

[PF79] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, 1979.

[PPRY18] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *FOCS*, 2018.

[PPY19] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. What storage access privacy is achievable with small overhead? In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS*, pages 182–199, 2019.

[PY19] Giuseppe Persiano and Kevin Yeo. Lower bounds for differentially private RAMs. In *Advances in Cryptology - EUROCRYPT 2019*, pages 404–434, 2019.

[RYF+13] Ling Ren, Xiangyao Yu, Christopher W. Fletcher, Marten van Dijk, and Srinivas Devadas. Design space exploration and optimization of path oblivious RAM in secure processors. In *The 40th Annual International Symposium on Computer Architecture, ISCA*, pages 571–582. ACM, 2013.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Advances in Cryptology - ASIACRYPT*, pages 197–214, 2011.

[SS13] Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious cloud storage. In *2013 IEEE Symposium on Security and Privacy*, pages 253–267. IEEE, 2013.

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *19th Annual Network and Distributed System Security Symposium, NDSS*, 2012.

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

[WCS15] Xiao Wang, T.-H. Hubert Chan, and Elaine Shi. Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 850–861, 2015.

[WHC+14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 191–202, 2014.

[WNL+14] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T.-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. In *ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 215–226, 2014.

[WST12] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: A parallel oblivious file system. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[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 *IEEE Symposium on Security and Privacy, S&P*, pages 218–234, 2016.

# A Sampling Noise on a Turing machine

One of the operations our differentially oblivious Turing machine needs to do is to sample from the (continuous) Laplacian distribution $\mathsf{Lap}(x)$; this is used in the algorithm for estimating the head's location; see Section 3. There, we need to generate a sample from $\mathsf{Lap}((1+\log_2 N)/\epsilon)$ and we need to do this about $N$ times. Recall that a Laplacian distribution is unbounded and samples need infinite precision. We show that with small tolerable loss in precision (which does not affect our final result), one can sample an approximation from this distribution on a standard Turing machine.

We assume that $\ln(1/\epsilon)$ is an integer so that we do not have rounding issues. Also, recall that $\delta$ is a negligible function of the form $\exp(-\log^2 N)$). First, we switch to a bounded version of the distribution, chopping off the tail which contains elements that occur with negligible probability. We can assume that we sample from the range $\pm(\log(N)/\epsilon)\cdot\mathsf{poly}\log(1/\delta)$. Let us call $\delta_0$ the probability mass that we chopped off. Sampling from the bounded version turns our $(\epsilon,\delta')$-differentially private prefix sum algorithm into an $(\epsilon,\delta)$-differentially private one, where $\delta=\delta'+N\cdot(e^\epsilon\cdot\delta_0+\delta_0)$. To see this, observe that we have essentially $N$ instances of the $\mathsf{Lap}$ noise and so by a simple union bound, the statistical distance between each event w.r.t the bounded distribution happens with probability at most $N\cdot\delta_0$ larger than in the unbounded version. Namely, for any set $S$, $\Pr[X_{\mathsf{bounded}}\in S]\le\Pr[X\in S]+N\cdot\delta_0$, where $X_{\mathsf{bounded}}$ is the output of the mechanism when using bounded noise and $X$ is the original mechanism. Then, by differential privacy, $\Pr[X\in S]+N\cdot\delta_0\le e^\epsilon\Pr[Y\in S]+\delta'+N\delta_0$, where $Y$ is another arbitrary event sampled from the unbounded noise version. Then again by bounding the noise used in $Y$, we get

$$\Pr[X_{\mathsf{bounded}}\in S]\le e^\epsilon(\Pr[Y_{\mathsf{bounded}}\in S]+N\delta_0)+\delta'+N\delta_0.$$

Since we think of $\delta_0$ as being negligible in $N$ and $\epsilon$ being a constant, $\delta$ is also negligible.[4]

[4]The above analysis was very loose. In particular, one can do a tighter analysis and not lose the linear-in-$N$ factor in $\delta$ but for our purposes it does not matter since $\delta$ is negligible in $N$ anyway.

The next step is to represent each element in the bounded range with finite precision. We want to lose at most $\delta$ factor in precision (which will add another additive $\delta$ factor to our additive error), and so if we use $\ell$ bits of precision, we have the inequality:

$$2^{-\ell} \cdot ((\log N)/\epsilon) \cdot \mathsf{poly}\log(1/\delta) \leq \delta.$$

This means that it is enough to use $\ell \in O(\log((\log N \cdot \log\log(1/\delta))/(\epsilon\delta))$ bits of precision which can be bounded by $O(\log^2(1/\delta))$ bits since $\delta$ is negligible in $N$ and $\epsilon$ is a constant. Therefore, all operations can be executed efficiently enough (in time $O(\mathsf{poly}\log N)$), which by slightly changing parameters (e.g., the value of $p$), does not affect our asymptotic upper bound on the running time of our differentially private Turing machine.