Electronic Colloquium on Computational Complexity, Revision 1 of Report No. 28 (2010)

Oblivious RAMs without Cryptographic Assumptions [1]

Miklós Ajtai
IBM Research, Almaden Research Center
ajtai@almaden.ibm.com

Abstract. We show that oblivious on-line simulation with only polylogarithmic increase in the time and space requirements is possible on a probabilistic (coin flipping) RAM without using any cryptographic assumptions. The simulation will fail with a negligible probability. If n memory locations are used, then the probability of failure is at most $n^{-\log n}$. Pippenger and Fischer has shown in 1979, see [11], that a Turing machine with one-dimensional tapes, performing a computation of length n can be simulated on-line by an oblivious Turing machine with two dimensional tapes, in time $O(n \log n)$, where a Turing machine is oblivious if the movements of it heads as a function of time are independent of its input. For RAMs the notion of obliviousness was defined by Goldreich in 1987 in [6], and he proved a simulation theorem about it. A RAM is oblivious if the distribution of its memory access pattern, which memory cells are accessed at which time, is independent of the program running on the RAM, provided that the time used by the program is fixed. That is, an adversary watching the memory access will not know anything about the program running on the machine apart from its total time. Ostrovsky, improving Goldreich's theorem, has shown in 1990, see [8], [9], [7], that a RAM using n memory cells can a be simulated by an oblivious RAM with a random oracle (where the random bits can be accessed repeatedly) so that the increase of the space and time requirement is only about a factor of $\mathrm{poly}(\log n)$ (Goldreich's factor was about $\exp[(\log n)^{1/2}]$). In both cases the oblivious RAM with a random oracle, can be replaced, by an oblivious probabilistic (coin-flipping) RAM, provided that we accept some unproven cryptographic assumptions, e.g., the existence of a one-way function. In this paper we show that simulation with an oblivious, coin-flipping RAM, with only a factor of $\mathrm{poly}(\log n)$ increase in time and space requirements, is possible, even without any cryptographic assumptions.

In the theorems of Goldreich and Ostrovsky it is assumed that the oblivious RAM has a protected CPU, that is, a constant number of registers, so that, the accesses to these registers are not included into the memory access pattern, and so they are not available to the adversary. We show that the protected CPU is not needed for the present, or former, results, since a protected CPU can be simulated on a RAM, so that the time requirement is increasing only by a constant factor, and the space requirement by an additive constant. In fact we may even let the adversary know which instruction is executed at each time, and the results still remain the same. (Note however, that the theorems of Goldreich and Ostrovsky hold even for an adversary who can see the contents of all memory cells outside the protected CPU. There is no equivalent statement in the present result.)

[1] An extended abstract of this paper will appear in the proceedings of STOC'2010

ISSN 1433-8092

The history of the problem. Pippenger and Fischer has shown in 1979, see [11], that a Turing machine with one-dimensional tapes performing a computation of length n can be simulated on-line by an oblivious Turing machine with two dimensional tapes in time $O(n \log n)$, where a Turing machine is oblivious if the movements of its heads as a function of time are independent of its output. Obliviousness means in other words that if an adversary is watching an oblivious Turing machine but can see only the movements of the heads without seeing the contents of the cells of the tapes, then, apart from n, the total time of computation on the original machine, he will not gain any information about the input.

For RAM machines obliviousness was defined by Goldreich, see [6] or [7] as a natural extension of this concept as we will describe later. Theorems which can be considered as the analogue of the mentioned theorems for Turing machine were proved by Goldreich and then improved by Ostrovsky, see [6], [8], [9], [7]. To state these result we use the RAM model as defined in [1]. Suppose that each memory cell of the machine contains a $0, 1$ sequence of length q. For the definition of obliviousness we divide the total memory into the CPU and the main memory. The CPU contains a constant number of cells (their number is independent of q) that we will call registers and the remaining memory cells form the memory of the machine. The registers has the following roles, the results of the arithmetic operations always appear in a register, the instruction pointer is a register, the read instruction puts the content of a memory cell into a register etc. Apart from that, there may be a few registers for temporary storage.

The motivation of the definition of obliviousness is the assumption that there is an adversary who does not see at all what happens in the CPU, but when the machine accesses a memory cell to fetch its content to put it into a register or to store the content of a register in a memory cell, then the adversary will know which memory cell was accessed. That is, the adversary knows the access pattern of the memory cells but does not have any information about the accesses of registers and instructions performed on their contents. Since the total time spent on a program cannot be hidden efficiently, we assume that it is known to the adversary. Therefore, in the following discussion, we will assume that the total time spent by the program is fixed. This picture was partly motivated by the problem of software protection, where a physically protected CPU is a natural and useful assumption, see [6], [8], [9],[7].

The RAM is oblivious in a deterministic sense if the access pattern of the memory cells (outside the CPU) is independent of the program that is executed. To make an oblivious RAM, in this deterministic sense, one solution is the following. At each time when an access is needed, the CPU reads and rewrites the contents of all of the memory cells used by the program, and this way it hides the single actual operation. There is no more efficient solution in this deterministic sense. Goldreich proved however, see [6], [7], that if the RAM has access to a random oracle then a much more efficient solution is possible. Namely, we will say that a program is oblivious if the distribution of its memory access pattern is independent of the of the program that is executed. Therefore the adversary does not gain any information about the program apart from the time used by it. In Goldreich's solution the time for executing the program increases by a factor of $2^{\sqrt{2 \log n \log \log n}}$, where the original program is using a RAM with n memory locations. (The oblivious simulation is on-line in the sense that the simulating program gets the input words and produces the output words in the same overall order than the original program.)

Ostrovsky has improved the efficiency of the simulation so that the time requirement of the oblivious simulation is increased only by a factor of $\mathrm{poly}(\log n)$ and in fact if the time t required by the machine is smaller than n, by a factor of $\mathrm{poly}(\log t)$, and the space requirement of the simulation is larger than the space requirement of the original program by the same $\min\{\mathrm{poly}(\log n), \mathrm{poly}(\log t)\}$ factor. See [8], [9], [7]. Ostrovsky's simulation is called the hierarchical algorithm because of the nature of the data structure used by the RAM. The hierarchical algorithm is also using a random oracle. For a more practical solution (in both cases) the random oracle may be substituted by a probabilistic (coin flipping) RAM, provided that we accept some unproven cryptographic assumption, in the case of Ostrovsky's theorem the existence of a one-way function. Note that a machine using a random oracle has the advantage compared to a coin flipping machine, that the random oracle remembers "for free" previous random steps, while a coin flipping machine, if it does not store this information, then it cannot recover it later. We also note that in the mentioned theorems the simulation can be performed with probability one.

Oblivious simulations are also of interest as a possible defense against so called "cache attacks", see Osvik, Shamir, and Tromer [10], and in general keeping computation more secure.

The main result. We show that oblivious on-line simulation with polylogarithmic increase in the time and space is possible on a probabilistic (coin flipping) RAM without using any cryptographic assumptions. The simulation will fail with a negligible probability. If n memory locations are used, then the probability of failure is at most $n^{-\log n}$. In case of failure the adversary will get some information about the input. The solution is an extension of Ostrovsky's hierarchical algorithm, some parts of the proofs remain unchanged, but there are also new difficulties leading to combinatorial problems. The most important change is that the randomization is done in different way, so that the machine is now able to recover efficiently the results of previously performed randomizations. This makes it possible to replace the random oracle by coin flipping.

Another improvement is that the assumption about the existence of a protected CPU can be dropped, we formulate the theorem directly on a RAM, that is, the adversary will also know which registers are accessed at each time, and which instruction has been executed. However for the proof we use the same protected CPU computational model as Goldreich and Ostrovsky and then show, that in a RAM, a protected CPU can be simulated.

While preparing this manuscript the author has been informed of a recent independent work about the same problem by Damgård, Meldgaard, and Nielsen, see [5].

Computational Model. We use a von Nuemann type random access machine, where data and program are not distinguished. For the definition for such a machine see e.g., in [1] the random access stored program (RASP) machines, in the modified form where the contents of the memory cells are not arbitrary integers, but integers in the interval $[0, 2^q - 1]$.

In order to present the main result concisely, without too many defintions, we restrict our attention to one specific RAM described in [1], with the mentioned modifications. However our results remain true for a very large class of machines including essentially every reasonable defintion for a RAM. Later we will discuss what is "reasonable" in more detail.

For each positive integer $q \geq 10$, $\mathcal{M}_q$ is a machine with 2^q memory cells each containing a sequence of $0, 1$ bits of length q. These cells will be called $\texttt{cell}(0), \texttt{cell}(1), ..., \texttt{cell}(2^q - 1)$. We will consider the sequences contained in the cells as the binary representations of natural numbers from the interval $[0, 2^q - 1]$, so if we say that a cell contains the natural number i, we mean it contains its binary representation. (We may restrict the number of memory cells if needed, by simply saying that a particular program is using only the first m cells for some $m < 2^q$. Therefore our requirement that the machine has $2^q - 1$ memory cells, is not a real restriction). The state of the machine at each time is a function which assigns to each of memory cells its content.

$\mathcal{M}_q$ has γ_0 instructions. The names or encodings of these instructions are integers in $[0, 2^q - 1]$.

$\texttt{cell}(0)$ is called the accumulator of the machine and $\texttt{cell}(1)$ the instruction pointer. (The choice of these particular cells for the mentioned roles have no significance.)

The machine $\mathcal{M}_q$ has six types of instructions. (a) arithmetic operations, (b) an instruction to generate a random number, (c) instructions moving data between the memory cells, (d) control transfer instructions, which determine which instruction will be executed next, (e) input/output instructions, and (f) the halt instruction to terminate the execution of the program. We will define each of these types later in more detail.

The machine $\mathcal{M}_q$ is working in cycles, each cycle counts as one time unit. In each cycle it does the following. It checks the content of the instruction pointer. Its content is interpreted as an address, of a memory cell, say, number i. Then the machine executes the instruction whose name is in cell i. An instruction may have parameters (for the sake of simplicity we assume that each instruction has at most one parameter). A parameter typically is the address of a memory cell. The content of cell $i + 1$ is considered as the parameter of the instruction in cell number i. The machine executes the instruction with the indicated parameter and then, if it is not a control transfer instruction, it increases the content of the instruction pointer by 2. If it is a control transfer instruction then the instruction defines the new content of the instruction pointer. We will say that *a memory cell or register is involved in an instruction* if its content is used by the machine to execute the instruction, or the result of the instruction is placed in it. We will use this concept by considering an adversary who wants to get some information about what the machine is doing, and at each instruction knows which memory cells/registers are involved in the instructions, but does not know their contents.

(a) The arithmetic instructions are $+, \times, -, \lfloor x/y \rfloor$, the constants $0, 1$, and $2^q - 1$. In case of the arithmetic operations of the form $f(x, y)$, at the time when the machine reads the instruction, which is in $\texttt{cell}(i)$, x must be in the accumulator, and y must be in the memory cell whose address is the parameter of the instruction, that is, y is in $\texttt{cell}(a)$ where a is the content of $\texttt{cell}(i+1)$. The result appears in the accumulator. In the case of the constants, the result of the instruction, that is, the constant, appears in the accumulator (and the value of the parameter is irrelevant).

(b) an instruction to generate a random number. A random integer from the interval $[0, 2^q - 1]$ appears in the accumulator, the value of the parameter is irrelevant.

(c) instructions moving data between the memory cells. Read instruction: if the value of the parameter is a, then the instruction puts the content of $\texttt{cell}(a)$ into the accumulator. Write instruction: if the value of the parameter is a, then the instruction puts the content of the accumulator into $\texttt{cell}(a)$.

(d) control transfer instructions, GOTO X instruction. If the value of the parameter is X then the content of the intruction pointer is changed into a. "IF $X = 0$ THEN GOTO Y" instruction. If the content of the accumulator is 0 then the content of the instruction pointer is changed into Y, where Y is the value of the parameter, otherwise the value of the instruction pointer is increased by 2. "IF $X > 0$ THEN GOTO Y" If the content of the accumulator is greater than 0, then the content of the instruction pointer is changed into Y, where Y is the value of the parameter, otherwise the value of the instruction pointer is increased by 2.

(e) input/output instructions. INPUT instruction. The input is written in the accumulator. OUTPUT instruction. The value of the accumulator is given as output. In both cases the value of the parameter is irrelevant.

(f) HALT instruction. Terminates the execution of the program.

The first few memory cells sometimes will be also called registers. (Intuitively this corresponds to the CPU of a computer.)

A state of the machine $\mathcal{M}_q$, as we have indicated earlier, is a function which assigns to each of it cells a possible content. A history of the machine $\mathcal{M}_q$ is a function defined on an inititial segment I of the natural numbers which assigns a state $S(t)$ to each $t \in I$ with the following property. Suppose that $t, t+1 \in I$ and in $S(t)$ the content of the instruction pointer is a. If a is the name of an instruction different of the input instruction and the random number generator instruction, then we get $S(t+1)$ from $S(t)$, by executing the instruction a with the values contained in the memory cells of $\mathcal{M}_q$ defined by $S(t)$. If a is the input or random number generator instruction then $S(t+1)$ must be a state that is obtained from $S(t)$ by executing instruction a with the values of the memory cells described in $S(t)$, and with a suitably chosen value of the input or the generated random number.

Before the machine starts to work, a program P_0 of constant length is placed in the memory. We will call this the starting program. We may think that this is a small program, whose role is to write in the memory a larger program P and data for P. When we will consider an adversary who wants to get some information about what the machine is doing, we will assume that P_0 is known to the adversary. Because of this, the exact way as P_0 gets into the machine is irrelevant.

We assume that a deterministic program P can run on the machine in the following way. (The assumption that P is deterministic is not essential, we make it only for the sake of simplicity.) Program P_0, the starting program, is already in the machine and asks for inputs. The first part of the input received by P_0 is the program P, (when we say program P, some data for the program may be included in it). P_0 writes the program P into the memory and then transfers the control to program P. While P is running it may ask for input and may also provide output. In this situation we will say that the program P_0 runs the program P. We assume that program P does not use all of the 2^q, q-bit words as possible outputs, say, P can use a q bit word in the output only if its last bit is 0. The remaining q-bit words will be called *exceptional words*, we will explain their roles later. (Actually only a single exceptional word will be needed.)

We will also consider cases when the starting program is not P_0 but another program P_1. P_1 will not run P in the sense described above, rather it will only simulate P in a sense that we will define later.

In the cases of both P_0 and P_1, we define the *history* of the machine $\mathcal{M}_q$, as the sequence of its states. The initial state of the machine $\mathcal{M}_q$ is the state when it starts to work. (E.g. it may contain the program P_0 or P_1 in its initial sate.) Further states of the machine are uniquely determined by its initial sate, by its inputs, and the results of the randomizations. A history of the machine $\mathcal{M}_q$ is a sequence H_t, $t = 0, 1, ...$, where H_t describes the state of the machine at time t, including whether it gives an output or gets an input and if yes what is its value.

H_0 is the initial state of the machine. If in the initial state the machine contains the program P_i and later it gets the program P and a sequence a as input, where a is intended as input for P, then the corresponding history will be denoted by $\texttt{history}(P_i, P, a)$. $\texttt{history}(P_1, P, a)$ is a random variable, with respect to the random steps of P_1, whose distribution will be denoted by $\Lambda(P_1, P, a)$.

The *visible history* of the machine $\mathcal{M}_q$ is a sequence $\mathcal{V}_t$, $t = 0, 1, ...$, where $\mathcal{V}_t$ consists of the following information, encoded in some way.

(a) the name of the instruction that was executed at time t,

(b) the addresses of the memory cells that were involved in the instruction executed at time t, together with their roles in the instruction, and the name of the intruction.

(c) if an exceptional output was given by the machine at time t, then this fact.

$\Lambda'(P_1, P, a)$ will denote the distribution on the visible histories induced by $\Lambda(P_1, P, a)$. The expression "visible history" is motivated by the assumption that this is what an adversary can see form the operation of the machine.

The *input sequence* of a history H of the machine is a sequence $a_0, a_{1,}...$ containing all of the inputs occurring in the history H in their order of occurrences. (The timing of the inputs is not included in this sequence.) The sequence $t_0, t_1, ...$, where t_i is the time when the machine gets input a_i, for $i = 0, 1, ...$ will be called the *input-time sequence* of history H. We define the analogue notions for output sequences as well. We get the input/output or *i/o sequence* of history H, by merging its input and output sequences in the order of their timing, and attaching to each element of the sequence an extra bit which tells whether the corresponding element represents an input or an output. We define *i/o-time sequence* of history H in a similar way, by merging the corresponding input-time an output-time sequences with the extra bits attached.

We will assume that each program P declares its *memory requirements*, that is, a natural number n with the property that P during its execution will use only the first n memory cells of $\mathcal{M}_q$. We assume that the number n is known to P_0, since it it is encoded in some way in the program P. (This is not an essential requirement it can be avoided in the same way as it is done in [8], [9] or [7], but for the sake of simplicity we leave this requirement in the formulation of our results.)

The *conceded history* of program P at input a will be a triplet $\langle n, T, \iota \rangle$, where (a) n is the memory requirement of P, (b) T is the total time used by $\mathcal{M}_q$, when executing P with input a from the start of P_0 till the machine halts, and (c) ι is the i/o-time sequence of $\texttt{history}(P_0, P, a)$. Intuitively, the conceded history is the part of the history, which cannot be hidden from the adversary, at least not without great loss of efficiency, and so we rather assume that the adversary already knows it. Therefore we may assume that the conceded history is fixed and consider the distribution of the visible history with the condition that the conceded history is what we fixed.

Assume now that we replace P_0 by another program P_1 which does not run P only it tries to simulate it in some way. In this case we also allow, that at an arbitrary time, P_1 *declares failure* by giving an exceptional output. (Intuitively this means that P_1 says that "I cannot continue the simulation in the required way". This may happen if the randomizations, performed by P_1, created an unlikely situation, where P_1 would need extra time to keep its data in working order, but this extra time would give information to an adversary about the program P or its input.) When failure is declared, $\mathcal{M}_q$ gives an exceptional output and then stops. Therefore the history and even the visible history of the machine will always show if failure was declared and at what time.

We will say that P_1 *is a correct simulating program*, if for all program P and for all input sequence a, $\texttt{history}(P_0, P, a)$ and $\texttt{history}(P_1, P, a)$ has the same input/output sequence with probability 1, with respect to the randomization done by P_1.

Consider now the situation when the input/output sequence of $\texttt{history}(P_1, P, a)$ may contain an exceptional output (that is, a declaration of failure). The maximal initial segment of the input/output sequence of $\texttt{history}(P_1, P, a)$ which does not contain a failure declaration will be called the *reliable i/o sequence of* $\texttt{history}(P_1, P, a)$.

The program P_1 *is a partially correct simulating program* if for all program P and for all input sequence a, the reliable i/o sequence of $\texttt{history}(P_1, P, a)$ is identical to an initial segment of the i/o sequence of $\texttt{history}(P_0, P, a)$ with probability 1, with respect to the randomization done by P_1.

Assume that $\varepsilon(n)$ is a function, so that for all natural number n, $\varepsilon(n)$ is a real in $[0, 1]$. We will say that *the program P_1 is an oblivious simulating program with an $\varepsilon(n)$ failure rate*, if the following conditions are satisfied:

(i) P_1 is a partially correct simulating program,

(ii) for all programs P and input sequences a, if $\mathcal{V}$ is a random visible history according to distribution $\Lambda'(P_1, P, a)$, then for each natural number t, the probability, that in $\mathcal{V}$ failure is declared at time t, is at most $\varepsilon(n)$, where n is the space requirement of P.

(iii) for each conceded history $\mathbf{h}$, there exists a distribution $\mathcal{D}_\mathbf{h}$ on the set of visible histories so that the following holds. For all program P and input sequence a, if $\mathbf{h}$ is the conceded history of $\texttt{history}(P_0, P, a)$, then the distribution $\mathcal{D}_\mathbf{h}$ is identical to the conditional distribution of $\Lambda'(P_1, P, a)$, with the condition that failure has not been declared. (That is, the distribution of the visible history of $\texttt{history}(P_1, P, a)$, with the condition that there is no failure, is uniquely determined by the conceded history.)

A simulating program as defined above, with high probability, will not give any information about P and a to the adversary, who knows only the visible history, in addition to what is already contained in the conceded history. Indeed, if the conceded history is fixed, then the distribution of the visible history is also fixed. So the adversary only sees a random value of a random variable whose distribution is known to him. Since the randomization is done by the coin flipping of $\mathcal{M}_q$, the result of it does not contain any additional information about P or a.

Suppose now that the total time in $\texttt{history}(P_0, P, a)$ is T. (T can be determined from the conceded history of $\texttt{history}(P_0, P, a)$.) Assume also that the total time required for the simulation is always at most T', that is, in $\texttt{history}(P_1, P, a)$ there is always a halt instruction no later then time T'. We claim that under these circumstances if P_1 is an oblivious simulating program with failure rate $\varepsilon(n)$, and we choose the pair $\langle P, a\rangle$ at random, so that $\langle P, a\rangle = \langle P^{(i)}, a^{(i)}\rangle$ for $i = 0, 1$ with probabilities $\frac{1}{2}$, where the conceded history of $\texttt{history}(P_0, P^{(i)}, a^{(i)})$ is the same $\mathbf{h}$ for $i = 0, 1$, then an adversary, knowing the visible history of $\texttt{history}(P_1, P, a)$, cannot decide whether $\langle P, a\rangle = \langle P^{(0)}, a^{(0)}\rangle$ or $\langle P, a\rangle = \langle P^{(1)}, a^{(1)}\rangle$ with a greater probability than $\frac{1}{2} + T'\varepsilon(n)$. Indeed, if S is a probabilistic strategy for the adversary, then we may consider the probability of its success both with the condition that there is a failure of the simulation, and with the condition that there is no failure. In the second case the conditional distribution of the visible history of $\texttt{history}(P_1, P^{(i)}, a^{(i)})$ is $\mathcal{D}_\mathbf{h}$ for both $i = 0$ and $i = 1$. Because of the symmetry the probability that S gives the correct answer is exactly $\frac{1}{2}$. Therefore by Bayes' theorem, even if in the case of failure S always gives the correct answer, the probability of success for S, without any conditions, will be at most $(1 - T'\varepsilon(n))\frac{1}{2} + T'\varepsilon(n) \le \frac{1}{2}(1 + T'\varepsilon(n))$.

Theorem 1 *There exists $c_1, c_2, c_3 > 0$, such that for all $q > 10$ there exists a program P_1 of length at most c_3 for the machine $\mathcal{M}_q$, so that for all sufficiently large q, P_1 is an oblivious simulating program with failure rate at most $n^{-\log n}$. Assume further that P is a program with input "a" so that the space requirement of* $\texttt{history}(P_0, P, a)$ *is n, and its total time is t. Then the space requirement of* $\texttt{history}(P_1, P, a)$ *is at most $n(\log n)^{c_1}$ and the total time of* $\texttt{history}(P_1, P, a)$ *is at most $t(\log n)^{c_2}$.*

Sketch of the proof of Theorem 1. First we formulate Lemma 1 below, that will show that the existence of a protected CPU in the sense of [6], [8], [9], and [7] is not needed as an additional assumption, since such a protected CPU can be simulated in $\mathcal{M}_q$.

We have defined the history of $\mathcal{M}_q$ for the case, when at time 0 the program P_0 or P_1 is in the memory of the machine, and it runs or simulates another program P, which may also get some input a. Therefore, the history depended on three parameters P_i, P, and a. Below, we will need the notion of history in a simpler case, when at time 0 the machine contains a program Q, and while this program is running, there is no further input. The history and visible history for this case is defined in the same way as earlier. We need this notion when we will consider a time interval $[t_1, t_2]$ from a history $\texttt{history}(P_i, P, a)$, with the property that there is no input in the interval $[t_1, t_2]$. The content of the memory of $\mathcal{M}_q$, at time t_1, can be viewed as a program Q which is running on the machine till time t_2. For discussing the program Q in itself only, we measure the time in a different way, that is, we say that Q starts at time 0 and works till time $t_2 - t_1$. The following general definition of a program is motivated by these considerations.

Definition. 1. A program P is a sequence of natural numbers in the interval $[0, 2^q]$. When we say that the machine $\mathcal{M}_q$ starts to work with P in its memory, then we mean that P occupies an initial segment of the memory cells, and the contents of the remaining cells are 0s. If P is deterministic and in the time interval $[0, t]$ the program P does not ask for an input, then P uniquely determines the sequence of the states of $\mathcal{M}_q$ in this interval, which is called the history of the program P in the time interval $[0, t]$, and is denoted by $\texttt{history}_{[0,t]}(P)$. The visible history in the same interval is the sequence whose ith element is the name of the instruction which is executed by P at time i, together with the memory cells involved in it. (The earlier defintion of "visible history" have had a larger number of parameters, so there is no danger that two notion of "visible history" could be confused.) $\texttt{length}(P)$ will denote the length of the program P, that is, the number of elements of the sequence P.

2. $\texttt{cont}_t^{(H)}(i)$ will denote the content of memory cell number i in history H at time t. If the choice of history is clear from the context we may omit the subscript $^{(H)}$. If P, Q are programs then $P \circ Q$ will denote their concatenation as sequences.

3. Suppose that P is a program, t, i, j are nonnegative integers $i \le j$. Then $\texttt{state}(P, t, i, j)$ is a sequence of q bit words defined in the following way. Assume that $\mathcal{M}_q$ starts to work from the initial state P, and works till time t without asking for input. Then $\texttt{state}(P, t, i, j)$ is the sequence $\texttt{cont}_t(i), ..., \texttt{cont}_t(j)$.

4. The set of memory cells $\texttt{cell}(0), ..., \texttt{cell}(i-1)$ will be denoted by $\texttt{cellset}(i)$.

Lemma 1 *There exists a $\beta > 0$, and for for all positive integers α there exists a $c_\alpha > 0$, such that for all $q > 10$ there exists a deterministic program P' for the machine $\mathcal{M}_q$ with $\texttt{length}(P') = \beta$ so that the following holds. For all deterministic programs P with $\texttt{length}(P) \leq \alpha$ and for all positive integers t, statement* (1) *implies that both conditions* (2) *and conditions* (3) *are statisfied, where*

(1) *if $\mathcal{M}_q$ starts to work from the initial state P, then $\mathcal{M}_q$ does not ask for any inputs, does not give any outputs, in the interval $[0, t]$, and all of the memory cells involved in the instructions during the time interval $[0, t]$ are in the set $\texttt{cellset}(\alpha)$.*

(2) $\texttt{state}(P, t, 0, \alpha - 1) = \texttt{state}(P' \circ P, c_\alpha t, \beta, \beta + \alpha - 1)$

(3) *if H is the history of the machine $\mathcal{M}_q$ starting from the initial state $P' \circ P$, then the machine does not ask for any input during the time interval $[0, c_\alpha t]$ in history H, and the visible history of H in the interval $[0, c_\alpha t]$ does not depend on P.*

The lemma will be proved in the section "Simulating a protected *CPU*."

Remark. 1. This lemma implies that a program P of constant size, whose execution involves only those memory cells where the program is residing at the start, can be run by another program efficiently in an oblivious way. (If α is not constant, then we may lose the efficiency, since c_α can be as large as α.)

2. Using this lemma we will be able to accomplish the same task obliviously on a RAM, that can be accomplished by using a protected CPU, if our goal is to make the program oblivious. Note that in [6], [8], [9], [7] the protected CPU is used for other tasks as well. Namely, with the help of a protected *CPU* the contents of all other memory cells can be encrypted, and so the adversary can even see the (encrypted) contents of all of the memory cells outside the protected *CPU*, without gaining any meaningful knowledge about the input. In the present context however, when we consider the knowledge of the adversary in an information theoretic sense, without taking into account the limitation imposed by computation, such an encryption is not possible.

3. In the proof of this lemma to access the memory cells obliviously is not a problem, since at each step we may access all of them. The problematic instructions are the conditional instructions of the type "IF $X = 0$ THEN GOTO Y". An adversary who knows the visible history, knows e.g., where is the instruction pointer, may conclude from this whether $X = 0$ or not. Even, if the instruction pointer would not be available for the adversary, the problem would remain, since the next memory cell used after the conditional instruction may be different in the $X = 0$ and $X \neq 0$ cases.

4. We assumed that both P and P' are deterministic. We will be able to use the lemma for probabilistic programs too in the following way. The randomizations will take place at predetermined times and we will apply the lemma only for the time intervals between the randomizations. P will work as a deterministic program, separately in each of these intervals, and the previously chosen random values will be parts of the initial states of P in these intervals.

We will use the same solutions for input/output instructions and instructions involving cells not in $\texttt{cellset}(\alpha)$. All of these instructions will be executed only at predetermined time and the lemma will be applied only to the time interval between them.

Now we return to the sketch of the proof of Theorem 1. We are following the idea of using Interactive Turing Machines, as described in [6], [8], [9], or [7], for the simulation. We present it in an essentially equivalent but formally somewhat different computational model. The reason why we use a different setup is to make it easier to express our present goals. We do not assume that the reader is familiar with the notion of Interactive Turing Machines.

Suppose that $\tilde{\gamma} > 0$ is a constant that we will choose later. P_1 will simulate the program P in the following way. We cut the memory of $\mathcal{M}_q$ into two parts. The first part consists of the set of the first $\tilde{\gamma}$ memory cells, which we will call the CPU (and later will play the role of a protected CPU). The second part consists of the remaining memory cells whose set will be called the memory module. This partition will be used in the following way. $\texttt{history}(P_0, P, a)$ is a sequence $H_0, H_1, ...,$ where H_t is the state of the machine $\mathcal{M}_q$ at time t. P_1 will represent H_t with a data structure R_t stored in the memory module. During the simulation, when going from time t to time $t+1$, P_1 has to transform R_t into R_{t+1} on the average in $\text{poly}(\log n)$ time.

The change in the history $\texttt{history}(P_0, P, a)$ from H_t to H_{t+1} involves only the contents of a constant number of memory cells. Namely those memory cells which are involved in the execution of the instruction at time t. (This may include an input appearing in the accumulator.) Therefore P_1 will do the following. (a) Using the data structure R_t, it will determine the contents of the memory cells which are involved in the instructions at time t in $\texttt{history}(P_0, P, t)$, and copy them into the CPU. (b) In the CPU it will determine which memory cells are those whose content are changing from H_t to H_{t+1}, and what are those changes. Then (c) based on the results of the previous step it changes R_t into R_{t+1}.

The difficulty is that the representation R_t must be chosen in a way that all of these steps can be carried out obliviously in the sense of Theorem 1. Step (b) can be carried out by using Lemma 1. The computation done in the CPU with a constant number of registers, can be transformed into an oblivious computation using the lemma, if we allow that the number of registers is a larger constant. As we mentioned earlier all of the randomizations, input output instructions and accesses for memory cells outside the CPU will be done at predetermined times and we apply Lemma 1 only to the time intervals between them. Therefore the only problem is to represent the states of the machine in history $\texttt{history}(P_0, P_1, a)$ in a way that the necessary changes can be made obliviously.

Since the total memory requirement of $\texttt{history}(P_0, P_1, a)$ is n memory cells, the problem can be formulated in the following way. There are n variables $x_0, ..., x_{n-1}$ with initial values 0, whose values are sometimes changing. R_t has to represent somehow the current evaluation of these variables so that the change of the value of a variable can be carried out obliviously in $\text{poly}(\log n)$ time. Below we formulate this problem in a framework which makes its solution a special case of Theorem 1.

This special case of Theorem 1 formulated in Lemma 2 has the following form. It restricts the theorem to a single program P, and it also weakens the adversary. First we describe the program P which will be used in Lemma 2. Program P does the following. If an input arrives of the form $\langle a, v, 0 \rangle$ then P puts the integer v into memory cell with address $a + k_0$, where k_0 is a fixed integer, and gives the output v. If an input arrives of the form $\langle a, v, 1 \rangle$ then P reads the content of memory cell number $a + k_0$ and gives its value as output. (We assume that the initial content of each memory cell is 0)

If a is restricted to the interval $[0, n-1]$ then we can say that P is maintaining the values of the variables $x_0, ..., x_{n-1}$, with initial values 0, so that if an input of the type $\langle a, v, 0 \rangle$ arrives a new value v is set for variable x_a, while an input $\langle a, v, 1 \rangle$ is interpreted as a question of the present value of variable x_a and P responds according to this. (In the first case the output v has no function, and in the second case the input v has no function. They are included only to make the lengths of the outputs and inputs the same in the two cases. This way the two cases have identical conceded histories, and so an adversary knowing only this cannot tell, whether in a specific instance the value of a variable was set, or a question was asked about it.) The triplet $\langle a, v, \delta \rangle$, $\delta = 0, 1$ may be encoded in several actual input words. The actual encoding is irrelevant the only important point is that the number of input words which code $\langle a, v, \delta \rangle$ is always the same. Assume that the values v are always integers in the interval $[0, \wp]$. A program P which accomplishes the task described above, will be called a memory-maintenance program with n variables and with values in $[0, \wp]$. Suppose that $\wp < 2^q$ and $n < 2^q$, then the memory requirement of the memory maintenance program is $n + c$ memory cells, where c is a constant.

As we said earlier Lemma 2 is a special case of Theorem 1 in the sense that the adversary in the lemma will be weaker than in the theorem. In the description of this weaker adversary, for each natural number l, M_l will denote the set of all memory cells number i of $\mathcal{M}_q$ with $i \geq l$. We replace the notion visible history (in the definition of obliviousness) by the notion of k-visible history, where the k-visible history of $\mathcal{M}_q$ is the sequence $\mathcal{V}_t$, $t = 0, 1, ...$, where $\mathcal{V}_t$ consists of the following information encoded in some way.

(a) the name of the instruction that was executed at time t, provided that at least one of the memory cells involved in the instruction is in M_k.

(b) the addresses of those memory cells in M_k that were involved in the instruction executed at time t

(c) if an exceptional output was given by the machine at time t, then this fact

We define now the notion of an k-oblivious simulating program the same way as an oblivious simulating program, but we replace the notion of visible history everywhere with k-visible history. In particular $\mathcal{D}_O$ in properties (ii), (iii), and (iv) of the definition will be a distribution on k-visible histories, and in property (iv) we are speaking about the distribution of the k-visible history of $\texttt{history}(P_1, P, a)$.

In other words we have now an adversary who does not see what happens in the first k memory cells. We will use this in the case when k is a constant. The machine this way will work as if it had a protected CPU with k registers. (By Lemma 1, we will be able to simulate this situation so that an adversary even with the complete original powers, that has been defined for Theorem 1, will not get any additional information.)

Lemma 2 *There exists a constant $\tilde{\gamma} > 0$ so that Theorem 1 holds if P is a memory maintenance program, and we require of P_1 that it must be only a $\tilde{\gamma}$-oblivious simulating program.*

Before we sketch the proof of Lemma 2 we give some simple definitions which will be useful both in the sketch and the final proof.

Definition. We will assume that $n = 2^d$ is a power of 2. Part of the memory of the machine $\mathcal{M}_q$ (outside the protected cells) is partitioned into blocks of poly($\log n$) memory cells, each will be called a bucket. We have altogether $2n - 1$ buckets. The set of buckets is partitioned into $d + 1$ classes $\mathcal{B}_0, \mathcal{B}_1, ..., \mathcal{B}_d$ called buffers. Buffer $\mathcal{B}_i$ contains exactly 2^i buckets $\mathbf{b}_{i,0}, ..., \mathbf{b}_{i,2^i-1}$.

Definition. 1. A tree T is partially ordered set with a largest element 1_T and with the property that for all $a, b \in T$ if $a, b \in T$ are incomparable then they do not have a common lower bound in T.

Definition. We assume that d is a positive integer, $n = 2^d$, and T_n is a binary tree of depth d with $2n-1$ elements. The ith level of T will be denoted by $L_i^{(T_n)}$. We have that $|L_i^{(T_n)}| = 2^i$ for $i = 0, 1, ..., d$. We will omit the subscript $^{(T_n)}$ if its choice is clear from the context. Assume that $\tau \in T_n \backslash L_d$. The two successors of τ will be labelled by 0 and 1, and according to this denoted by $\tau^{(0)}$ and $\tau^{(1)}$. Suppose that $\delta = \langle \delta_0, ...\delta_{d-1} \rangle$ is a 0, 1 sequence of length d. We associate with the sequence δ a branch of the tree $b_0, ..., b_d$ with the property that $b_0 = 1_{T_n}$ and $b_{i+1} = b_i^{(\delta_i)}$ for all $i = 0, 1, ..., d-1$. Clearly $b_i \in L_i$. We will use the notation $\texttt{branch}(\delta) = \{b_0, ..., b_{d-1}\}$, $\texttt{node}(\delta, i) = b_i$ and $\texttt{leaf}(\delta) = b_{d-1}$. $\mathbf{L}_j = \mathbf{L}_j^{(T_n)}$ will denote the set $\bigcup_{s \le j} L_s^{(T_n)}$. The ancestor unique of an element $\tau \in T_n$ will be denoted by τ°.

Sketch of the proof of Lemma 2. The program P_1 of the lemma will maintain the values of the variables almost the same way as the "hierarchical algorithm" given by Ostrovsky, see [8], [9], [7]. The main difference will be in the randomization. Because of that, we add extra steps to the hierarchical algorithm as explained below.

First we sketch the basic idea of Ostrovsky's hierarchical algorithm in the present context. We make some minor changes compared to the original formulation, however these do not affect the substance.

Assume that the algorithm $\mathcal{H}$, using a random oracle, wants to store and recall the values of the variables $x_0, ..., x_{n-1}$ in an oblivious way. (The new algorithm with coin flipping only, will be denoted by $\mathcal{Y}$.) For the sake of simplicity and without the loss of generality we assume that $n = 2^d$, where d is an integer.

We will use the buckets and buffers defined earlier. In the case of algorithm $\mathcal{H}$, a bucket will contain $O(\log n)$ memory cells. The buckets will contain information about the values of the variables in form of pairs $\langle a, v \rangle$, where $a \in [0, n-1]$, $v \in [0, \wp]$. These pairs are represented in a way that a pair occupies only a constant number of memory cells. A bucket can contain only one copy of a pair.

While $\mathcal{H}$ is working, at each time, and for each $a \in [0, n-1]$, there is exactly one pair of the form $\langle a, v \rangle$ which is contained in one of the buckets, and it is contained only in one bucket. If the pair $\langle a, v \rangle$ is contained in a bucket, at a time, then the value of the variable x_a at this time is v. Therefore the goal of algorithm $\mathcal{H}$ is to keep these pairs in places that can be found in poly($\log n$) time on the average in an oblivious way. Algorithm $\mathcal{H}$, using a random oracle, accomplishes this in the following way.

Algorithm $\mathcal{H}$ will get, through its inputs, either a request for the value of a variable, or a request for changing its value. These requests will be handled in a similar way, therefore we call each of them a request for an access for the variable. The time of algorithm $\mathcal{H}$ (after an initial setup) is partitioned into intervals $\mathbf{A}_1, \mathbf{B}_1, \mathbf{A}_2, \mathbf{B}_2, ..., \mathbf{A}_i, \mathbf{B}_i, ...$ which are coming in the given order. The ith request for an access of a variable will arrive at the beginning of interval $\mathbf{A}_i$ and

$\mathcal{H}$ completes the necessary changes and gives the required output in this interval. Therefore we will call $\mathbf{A}_i$ an access interval. The length of the time interval $\mathbf{A}_i$ will be $O(\text{poly}(\log n))$, and it will be independent of i.

The intervals $\mathbf{B}_i$ are used for a random rearrangement of the pairs in the buckets. These intervals are called epochs in Ostrovsky's description of the hierarchical algorithm. Sometimes we will also call them bookkeeping intervals. The length of an epoch $\mathbf{B}_i$ will depend on i in a way that, for all $k = 1, 2, ...$ the total length of epochs $\mathbf{B}_1, ..., \mathbf{B}_k$ will not be greater than $k\text{poly}(\log n)$. We will achieve this by requiring that $|\mathbf{B}_i| \leq \text{poly}(\log n)2^{\mathbf{a}(i)}$, where $\mathbf{a}(i)$ is the largest integer so that $2^{\mathbf{a}(i)}$ is a divisor of both i and $n = 2^d$.

First we describe what happens in an interval $\mathbf{B}_i$. The algorithm $\mathcal{H}$ does not keep any other data then i and the pairs in the buckets. In epoch $\mathbf{B}_i$ the algorithm $\mathcal{H}$ reorganizes the contents of the buckets in the following way. $\mathcal{H}$ goes through all of the buckets in $\bigcup_{j \leq \mathbf{a}(i)} \mathcal{B}_j$ in a predetermined order. For each pair $\langle a, v \rangle$ that $\mathcal{H}$ finds in any of these buckets it randomly chooses a new destination bucket, where they have to go, by the end of $\mathbf{B}_i$. The randomization is done by the random oracle by providing a value $h(a, i) \in \{0, 1, ..., 2^{\mathbf{a}(i)} - 1\}$ with uniform distribution. The meaning of the number $h(a, i)$ is that the pair $\langle a, v \rangle$ must get into bucket $\mathbf{b}_{\mathbf{a}(i),h(a,i)}$ of buffer $\mathcal{B}_{\mathbf{a}(i)}$ by the end of epoch $\mathbf{B}_i$. $\mathbf{b}_{\mathbf{a}(i),h(a,i)}$ will be called the destination bucket of the pair $\langle a, v \rangle$. It is important that the random hash value $h(a, i)$ depends only on a and i. At this time only the random choice of the destination bucket takes place, so that each of the destination buckets has uniform distribution on buffer $\mathcal{B}_{\mathbf{a}(i)}$ and all of these choices are mutually independent. $\mathcal{H}$ does not take out the pairs at this point from the buckets, it only attaches the random destination to each pair, some way.

Since the buckets contain only $O(\log n)$ memory cells, this choice of and recording of destinations can be done obliviously using poly$(\log n)$ time for each bucket that is altogether $2^{\mathbf{a}(i)}\text{poly}(\log n)$ time.

The next phase of the algorithm, whose details are not sketched here, takes each pair to its chosen destination. This is done by using several times, an oblivious sorting network e.g., Batcher's algorithm or AKS network. A possible way to do this is described in [8], [9], and a slightly different one in [7]. We will use this technique, with either of the two solutions, in the present proof without any essential changes. For our purposes, it is enough to know, that each pair gets into its chosen destination in a deterministic and oblivious way in time $2^{\mathbf{a}(i)}\text{poly}(\log n)$. This completes the description of $\mathcal{H}$ in the epoch $\mathbf{B}_i$.

We note here that with a small probability it may happen that too many pairs has the same destination bucket and, so there is not enough memory cells in the bucket to store all of the pairs sent there, We will say in such a case that the bucket is overfilled. In algorithm $\mathcal{H}$ this does not cause a serious problem, since if this happens, the whole randomization can be repeated and the additional expected time will be very small. However in the new algorithm $\mathcal{Y}$, using coin flipping only, this treatment of overfilling could leak information to the adversary. Therefore we will use in algorithm $\mathcal{Y}$ bigger buckets with size $O(\text{poly}(\log n))$, and will show that with high probability no buckets will be overfilled. (Since the randomization will be different this requirement will create major difficulties in the proof.) If a bucket is still overfilled then the algorithm $\mathcal{Y}$ will declare failure.

The described action of algorithm $\mathcal{H}$ in the epochs $\mathbf{B}_i$ has the result, that if a pair $\langle a, v \rangle$ is not moved during the intervals $\mathbf{A}_j$ then during the various epochs it travels through the buffers

$\mathcal{B}_0, \mathcal{B}_1, ..., \mathcal{B}_d$ in this order and then it stays in $\mathcal{B}_d$. Motivated by this we may define the initial setup in a way that all of the pairs $\langle a, 0\rangle$, $a \in [0, n-1]$ are put in the buckets of $\mathcal{B}_d$ in a random way. The randomization can be accomplished in a similar way, that we have described above. Actually this initial setup can be avoided and the number of buffers and buckets can be increased gradually putting only pairs $\langle a, v\rangle$ into them where there was already a request for access for the value of x_a, see [8], [9]. For the sake of a picture with less complicating factors for the coin flipping algorithm that we will describe later, we allow an initial setup.

Now we describe the algorithm $\mathcal{H}$ in an access interval $\mathbf{A}_j$. At the beginning of this interval a request arrives concerning the value of the variable x_{a_j}. The algorithm $\mathcal{H}$ has to find a pair of the form $\langle a_j, x\rangle$ which must be in one of the $2n-1$ buckets. Since the time allowed for this step is only poly$(\log n)$, $\mathcal{H}$ cannot look in all of the buckets. It searches for the pair in the following way. $\mathcal{H}$ searches for the bucket containing $\langle a_j, x\rangle$ in the buffers $\mathcal{B}_0, ..., \mathcal{B}_d$ in this order. When it gets to buffer $\mathcal{B}_r$, for some fixed $r = 0, 1, ..., d$, $\mathcal{H}$ does the following. Knowing r and j, it finds the largest integer $i_r < j$ so that $\mathbf{a}(i_r) = r$. The importance of this integer i_r is that $\mathbf{B}_{i_r}$ was the last epoch, when the destination buckets were in buffer $\mathcal{B}_{\mathbf{a}(i_r)} = \mathcal{B}_r$. Therefore, if at the beginning of $\mathbf{B}_{i_r}$ the pair $\langle a_j, x\rangle$ was in a bucket in $\bigcup_{s\leq r} \mathcal{B}_s$, then during $\mathbf{B}_{i_r}$ it got into a bucket in buffer $\mathcal{B}_r$, namely into bucket $\mathbf{b}_{r,h(a_j,i_r)}$. $\mathcal{H}$ using the random oracle determines $h(a_j, i_r)$ and checks whether the pair is really there. If $\mathcal{H}$ finds the pair it removes it and puts it into the single bucket $\mathbf{b}_{0,0}$ of buffer $\mathcal{B}_0$.

Even if $\mathcal{H}$ has found a pair $\langle a_j, x\rangle$ in one of the buffers, it repeats the same action for the remaining buffers pretending still looking for the pair $\langle a_j, x\rangle$ and pretending putting it into $\mathbf{b}_{0,0}$.

This action of $\mathcal{H}$ reveals for the adversary a bucket in each buffer $\mathcal{B}_r$ that is searched by $\mathcal{H}$. This bucket is $\mathbf{b}_{r,h(a_j,i_r)}$ in $\mathcal{B}_r$. For a fixed a_j and i_r, $h(a_j, i_r)$ has uniform distribution on $[0, 2^r - 1]$. Therefore at this point the adversary gets the value of $h(a_j, i_r)$. At other times during the execution of the algorithm $\mathcal{H}$ the adversary will get other values $h(u, v)$ but never again with $u = a_j$, $v = i_r$. Therefore when the adversary gets the values of h, he will simply see the values of independent random variables with predetermined distribution. Consequently the adversary does not gain any information from the addresses of the buckets searched by $\mathcal{H}$. The search of the buckets and all of the other actions of $\mathcal{H}$ can be done in a way that the sequence of these buckets uniquely determine the memory access pattern of $\mathcal{H}$.

After all of the buffers have been searched in the described way, the pair $\langle a_j, x\rangle$ is in the bucket of $\mathcal{B}_0$. Changing x, if needed, and giving the appropriate output can be done obliviously in a deterministic way. This describes the action of $\mathcal{H}$ in an access interval $\mathbf{A}_j$.

Summarizing this process, in each access interval we bring the pair $\langle a, x\rangle$ to the buffer $\mathcal{B}_0$, where x_a is the variable to be accessed, and x is its current value. Later, during the epochs $\mathbf{B}_j$, the pair $\langle a, y\rangle$, where y is the new value of the variable x_a will go through the buffers $\mathcal{B}_0, \mathcal{B}_1, ...$ with a random choice of the bucket where it is located, until there will be a request again concerning x_a. The algorithm is oblivious, because the only memory accesses which are not known in advance are the accesses for the contents of the buckets $\mathbf{b}_{r,h(a_j,i_r)}$, $i = 1, 2, ...,$ $r = 1, 2, ...d$ described above, and, as we have seen, this sequence has a predetermined distribution which is independent of the input.

Replacing the random oracle with coin flipping. We describe now an algorithm $\mathcal{Y}$, with coin flipping instead of using a random oracle, which essentially accomplishes the same thing as algorithm $\mathcal{H}$. Later we will modify algorithm $\mathcal{Y}$ because in the simple form presented here the probability of overfilling of a bucket may be too high. However this modified algorithm will differ from $\mathcal{Y}$ only in the choice of a parameter. The reason why the algorithm is working remains the same.

Suppose that $\mathcal{Y}'$ is the algorithm that we get from $\mathcal{H}$ by simply using coin flipping instead of a random oracle. The problem will be that algorithm $\mathcal{Y}'$ will not know what was the destination of the pair $\langle a_j, x\rangle$ when it was put into buffer $\mathcal{B}_{\mathbf{a}}(i) = \mathcal{B}_r$. To keep this information in the memory even for buffer $\mathcal{B}_1$ with only 2 buckets, would require keeping n bits of information in the memory so that they are available obliviously. This is a task almost as difficult as the original problem. Therefore algorithm $\mathcal{Y}'$ does not meet our requirements.

To overcome this difficulty our solution is the following. Instead of putting the pairs $\langle a, x\rangle$ directly into a bucket in the physical memory of $\mathcal{M}_q$, first we give them a symbolic address (may be also called a logical address) which will be a node of the binary tree T_n defined earlier. This symbolic address will depend only on a, and not on x, but it may change in each time interval $\mathbf{A}_i$ or $\mathbf{B}_i$, that is, it also depends on the time. The current symbolic address of an $a \in \{0, 1, ..., n-1\}$ will be denoted by $\mathtt{symb}(a)$. If $\mathtt{symb}(a) = \tau$, then the pair $\langle a, x\rangle$ is at node τ of the tree. This assignment of symbolic addresses will be done in a random way, as we will describe below, but using only $\mathrm{poly}(\log n)$ random bits, which can be kept in the memory so that the symbolic addresses can be recalculated d-obliviously any time.

Recall that L_i has denoted the ith level of the tree as defined in the definition of T_n. Separately for each node τ of the tree in L_i, $i = 0, 1, ..., d$ we will select a random bucket $\mathbf{p}(\tau)$ in $\mathcal{B}_i$ with uniform distribution. The pair $\langle a, x\rangle$ with $\mathtt{symb}(a) = \tau$ will be stored in bucket $\mathbf{p}(\tau)$. So we may say that the physical address of $\langle a, x\rangle$ will be $\mathbf{p}(\tau)$, if $\mathtt{symb}(a) = \tau$. $\mathcal{Y}$ will select $\mathbf{p}(\tau)$ with uniform distribution on $\mathcal{B}_i$, if $\tau \in L_i$. These selections will be mutually independent, and also independent of the random selections of the symbolic addresses $\mathtt{symb}(a)$. $\mathbf{p}(\tau)$ will change during the execution of algorithm $\mathcal{Y}$ but only in the intervals $\mathbf{B}_i$. Since $\mathcal{Y}$ has to know which is the bucket $\mathbf{p}(\tau)$, when it has to find a pair $\langle a, x\rangle$ with $\mathtt{symb}(a) = \tau$, the address of $\mathbf{p}(\tau)$ in $\mathcal{M}_q$, that we will denote by $\bar{\mathbf{p}}(\tau)$, must be stored somewhere, and when needed recovered in an oblivious way. For all possible $\tau \in T_n$ storing $\bar{\mathbf{p}}(\tau)$ obliviously requires altogether storing about $O(n \log n)$ bits obliviously, just like in the original problem. Our solution will be, as we describe in more details below, that we will store the addresses $\bar{\mathbf{p}}(\tau)$ in the buckets, the same way as we store the data about the values of the variables x_a. More precisely we will store pairs $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau)\rangle$, where the element τ of the tree will be represented by the integer $\bar{\tau}$, in some efficient way. To do that each node $\tau \in T_n$, $\tau \neq 1$ will have a symbolic address $\mathtt{symb}_T(\tau)$ which is an element of the tree with $\mathtt{symb}_T(\tau) > \tau$ and the pair $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau)\rangle$ will be stored in the bucket $\mathbf{p}(\mathtt{symb}_T(\tau))$. The requirement $\mathtt{symb}_T(\tau) > \tau$ is very important. This will guarantee that as $\mathcal{Y}$ will go down on a branch B of the tree starting from the root, and looking into all buckets $\mathbf{p}(\sigma)$ with $\sigma \in B$, by the time $\mathcal{Y}$ reaches a $\tau \in B$ and needs to know what is $\bar{\mathbf{p}}(\tau)$ it already have seen it in a bucket $\mathbf{p}(\sigma)$ for some $\sigma > \tau$. We will describe below more precisely how this is done by $\mathcal{Y}$. Since $\mathbf{p}(1_{T_n}) = \mathbf{b}_{0,0}$, we do not need a symbolic address for 1_{T_n}.

The algorithm $\mathcal{Y}$ generates a random function χ which assigns to each integer $a \in [0, n-1]$ a $0, 1$ sequence $\chi(a)$ of length d. We require from the function χ that for any subset $A \subseteq [0, n-1]$ with $|A| \le \mathrm{poly}(\log n)$ the random variables $\chi(a)$, $a \in A$ are mutually independent. Such a random function χ can be generated by selecting only $\mathrm{poly}(\log n)$ random bits. For example $\chi(a)$ can be the determined by the value of a random polynomial of degree $(\log n)^c$ over the finite field F_{2^d}. The coefficients are chosen uniformly from F_{2^d} and can be represented altogether by $(\log n)^{c+1}$ random bits. The value of the polynomial is an element of the field which over a fixed basis, can be interpreted as a $0,1$ sequence of length d. Therefore the random function χ can be generated by $\mathrm{poly}(\log n)$ coin flipping during the setup, the algorithm $\mathcal{Y}$ will keep the results of these coinflips, and so it can recalculate $\chi(a)$ d-obliviously any time for each $a \in [0, n-1]$. It is very important that *the randomization of* χ *will be performed only once* and the same $\mathrm{poly}(\log n)$ random bits defining χ are used till $\mathcal{Y}$ halts.

Recall that a branch was assigned to each $0,1$ sequence s of length d, and denoted by $\mathtt{branch}(s)$. Therefore by the randomization of χ we also assign a random branch $\mathtt{branch}(\chi(a))$ to each integer $a \in [0, n-1]$. For such an integer a, $\mathtt{symb}(a)$ *will be always in* $\mathtt{branch}(\chi(a))$.

$\mathtt{symb}(a)$ for $a \in [0, n-1]$ will be determined in the following way. At the beginning $\mathtt{symb}(a)$ is the leaf at the end of $\mathtt{branch}(\chi(a))$.

Suppose that in an interval $\mathbf{A}_j$ the input is $\langle a_j, v_j, \Psi_j \rangle$. Then for all $a \in [0, n-1]$, if $\mathtt{symb}(a) \in \mathtt{branch}(\chi(a_j))$ at the beginning of $\mathbf{A}_j$ then at the end of $\mathbf{A}_j$, $\mathtt{symb}(a)$ is changed into 1_{T_n}. The symbolic addresses of all of the other integers $a \in [0, n-1]$ remain unchanged.

In an epoch $\mathbf{B}_i$ the change is the following. If at the beginning of $\mathbf{B}_i$ we have $\tau = \mathtt{symb}(a) \in \bigcup_{s \geq \mathbf{a}(i)} L_s$ then $\mathtt{symb}(a)$ remains unchanged in $\mathbf{B}_i$. Otherwise, that is, if $\tau \in \bigcup_{s < \mathbf{a}(i)} L_s$, $\mathtt{symb}(a)$ is changed into the unique element of $\mathtt{branch}(\chi(a)) \cap L_{\mathbf{a}(i)}$. This completes the description of how the symbolic addresses are selected.

The selections of the buckets $\mathbf{p}(\tau)$. We have to assign to each node $\tau \in L_i$ a bucket $\mathbf{p}(\tau)$ in buffer $\mathcal{B}_i$ for $i = 0, 1, ..., d$. For each $\tau \in L_i$, $\mathbf{p}(\tau)$ will be chosen at random with uniform distribution from $\mathcal{B}_i$ and the choices will be mutually independent for $\tau \in L_i$, $i = 0, 1, ..., d$, moreover they will be also independent of the choice of the function χ. At the startup we make a random selection for $\mathbf{p}(\tau)$, for each $\tau \in T_n$, and update it only in the time intervals $\mathbf{B}_i$. In the access intervals $\mathbf{A}_i$ all physical addresses $\mathbf{p}(\tau)$ remain unchanged.

In the epoch $\mathbf{B}_i$ algorithm $\mathcal{Y}$ updates the assigned addresses only for the nodes in $\bigcup_{s \leq \mathbf{a}(i)} L_s$. $\mathcal{Y}$ randomizes a new bucket $\mathbf{p}(\tau)$ for each node $\tau \in L_s$ with uniform distribution on $\mathcal{B}_s$, for $s = 0, 1..., \mathbf{a}(i)$. As in the startup these randomization are mutually independent, and independent of all of the earlier randomizations. This completes the description of the selections of the buckets $\mathbf{p}(\tau)$. These randomizations are the analogues of the randomizations done by the hierarchical algorithm $\mathcal{H}$ in the interval $\mathbf{B}_i$ using the random oracle and will be used for the same purpose. In the present case however, the randomizations are done by coin flipping, so the algorithm $\mathcal{Y}$ have to remember the results in some way.

We describe now how the algorithm $\mathcal{Y}$ stores the results of the randomizations of the buckets $\mathbf{p}(\tau)$. The current $\bar{\mathbf{p}}(\tau)$, that is, the address of the bucket assigned to the node τ, will be stored in the form of a pair $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau) \rangle$. Such a pair will be placed in a bucket in the same way as we do with the pairs $\langle a, v \rangle$ storing the values of variables. Therefore τ will have symbolic address $\mathtt{symb}_T(\tau) \in T_n$ and the pair $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau) \rangle$ will be stored in the bucket assigned to $\mathtt{symb}_T(\tau)$, that is, in $\mathbf{p}(\mathtt{symb}_T(\tau))$.

There is however an important complication. Namely, for each $\tau \in T_n$ the algorithm $\mathcal{Y}$ can search the bucket $\mathbf{p}(\tau)$ only once, otherwise the obliviousness of the algorithm is lost. $\mathcal{Y}$ may search the same bucket as $\mathbf{p}(\sigma)$ for another node $\sigma \neq \tau$ of the tree, but in the role of $\mathbf{p}(\tau)$. The reason is that for the adversary a single search looks like searching in a randomly chosen bucket, while repeated searches in the same bucket may indicate e.g., repetition in the input. (This is the same situation as in [9] and [7].) The solution for $\mathcal{Y}$ will be that after it used the bucket $\mathbf{p}(\tau)$ once, it erases the pair $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau)\rangle$. So the address of $\mathbf{t}(\tau)$ is simply not available till the next randomization of $\mathbf{p}(\tau)$, when $\mathcal{Y}$ creates a new pair $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau)\rangle$.

Changes of the symbolic address of $\mathtt{symb}_T(\tau)$ *during the algorithm.* At the startup $\mathtt{symb}_T(\tau) = \tau^\circ$, where τ° is the unique ancestor of τ. This can be done only if $\tau \neq 1_T$, but for $\tau = 1_T$, $\mathbf{p}(\tau)$ is always $\mathbf{b}_{0,0}$, so $\mathcal{Y}$ does not have to store this information. Therefore, at the beginning, the pair $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau)\rangle$ will be stored in the bucket assigned to the node τ°, that is, in $\mathbf{p}(\tau^\circ)$. When we randomize a new $\mathbf{p}(\tau)$ then the old pair $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau)\rangle$ is erased from the memory.

In an access interval $\mathbf{A}_i$, with input $\langle a_i, v_i, \Psi_i\rangle$ the following happens. If $\mathtt{symb}_T(\tau) \in \mathtt{branch}(\chi(a_i))$ then $\mathtt{symb}_T(\tau)$ will change into 1_{T_n} at the end of $\mathbf{A}_i$, otherwise it remains unchanged.

We consider now the change of $\mathtt{symb}_T(\tau)$ in an epoch $\mathbf{B}_j$. If $\mathtt{symb}_T(\tau) \in \bigcup_{s \geq \mathbf{a}(i)} L_i$, then $\mathtt{symb}_T(\tau)$ remains unchanged in $\mathbf{B}_i$. Otherwise, at the end of $\mathbf{B}_i$ it changes into ρ, where ρ is the smallest node in $\bigcup_{s \leq \mathbf{a}(i)} L_i$ which is larger than τ. This completes the definition of the symbolic address of $\mathtt{symb}_T(\tau)$.

This definition implies that in the access interval $\mathbf{A}_j$ with input $\langle a_j, v_j, \Psi_j\rangle$, the algorithm $\mathcal{Y}$ will be able to go down the branch $\chi(a_j)$ starting from 1_T and check for each node τ the bucket $\mathbf{p}(\tau)$. Indeed, as we have seen, the pair $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau)\rangle$ is always contained in a bucket $\mathbf{p}(\rho)$, for some $\rho \geq \tau^\circ > \tau$, which is also on the branch, but comes earlier. Therefore when the algorithm $\mathcal{Y}$ goes down the branch $\mathtt{branch}(\chi(a))$, looking in the buckets assigned to the nodes, by the time it reaches the node τ it will know what is $\bar{\mathbf{p}}(\tau)$.

This completes the sketch of the coinflipping algorithm. The reason why it finds the pair $\langle a, v\rangle$ is the same as in the case of Ostrovsky's hierarchical algorithm. The reason why it will be oblivious is also the same. Namely once $\mathcal{Y}$ looks at a bucket $\mathbf{b}_{i,j}$, all of the symbolic addresses $\mathtt{symb}(a)$ and $\mathtt{symb}(\tau)$ whose value was $\mathbf{b}_{i,j}$ till now, are changed into 1_{T_n}. Therefore the bucket in the same role will not be used again by $\mathcal{Y}$. This implies that the buckets checked by $\mathcal{Y}$ are the values of independent random variables with previously fixed distributions. So the adversary does not gain any information from them. On the other hand if we fix the sequence of buckets that $\mathcal{Y}$ checks, then $\mathcal{Y}$ is oblivious in a deterministic sense, and therefore there is no other information available to the adversary than the bucket sequence.

Problems concerning the algorithm $\mathcal{Y}$*.* As we have remarked earlier overfilling a bucket if the algorithm is using an oracle is not a problem. For a detailed explanation see [8], [9] or [7]. The essential reason is that in case of an overfilling by $\mathcal{H}$ the randomization, which is a random hashing, can be repeated. $\mathcal{H}$ can be executed in a way that the fact that there is an overfilling gives only information about the randomization and not about the pairs $\langle a, v\rangle$ in the buckets.

This is not true for the algorithm $\mathcal{Y}$. The randomization of χ creates a new problem. This randomization cannot be repeated frequently, since such a repetition would involve relocating all of the pairs $\langle a, x\rangle$ into new bucket, so it would take at least $n \log n$ time. An overfilling may result, for example, from the event that for many of the arriving access requests $\langle a, v\rangle$, the set $\mathtt{branch}(\chi(a))$ is the same. Therefore, if χ is not re-randomized then repeating the other part of the randomization will not help. Consequently, to prove the theorem, we have to show that with a probability higher than $1 - n^{-\log n}$, no overfilling will occur.

To prove this, first we have to show that

(4) *for each fixed* $\tau \in T_n$ *the number of* $a \in [0, n-1]$ *with* $\texttt{symb}(a) = \tau$, *will be at most* $\mathrm{poly}(\log n)$.

(We will show that this implies that for each fixed $\tau \in T_n$ we have $|\{\sigma \in T_n \mid \texttt{symb}_T(\sigma) = \tau\}| \le \mathrm{poly}(\log n)$). The randomization of the physical addresses $\mathbf{p}(\tau)$ does not cause a serious problem, it will only slightly increase the exponent in $\mathrm{poly}(\log n)$ in the upper bound.

For the proof of the upper bound (4) we have to modify the algorithm $\mathcal{Y}$. (The upper bound may hold for the original form of $\mathcal{Y}$ as well, but the proof breaks down.) The disadvantage of the presented form of the algorithm $\mathcal{Y}$ is that the expected number of pairs $\langle a, x\rangle$ in each bucket $\mathbf{b}_{i,j}$ is a constant. Moreover, if we fix χ, and consider only the other part of the randomization, then the expected number of pairs in each fixed bucket, will depend on the choice of the bucket. For some buckets, this expected number can be as high as about $\log n$. On the other hand, for the proof, at least with the technique that we will use, we need this expected number (for buckets which do not correspond to the leaves of the tree) to be as low as $(\log n)^{-c_1}$, for some constant $c_1 > 0$, that will be selected later. The reason is that during the algorithm when we move one pair $\langle a, v\rangle$ to the bucket $\mathbf{b}_{0,0}$ we will have to move others pairs $\langle a', v'\rangle$ as well where $\texttt{symb}(a') \in \texttt{branch}(\chi(a))$. This may lead to the situation that the evaluation pairs are moving faster, or in larger numbers, to the top of the tree then they are moving downward towards the leaves of the tree during the epochs $\mathbf{B}_i$. One possible solution for the problem could be, that after each access interval $\mathbf{A}_i$, we will have not one but $(\log n)^{c_1}$ epochs $\mathbf{B}_i$. Actually we will use an equivalent solution, instead of having many epochs after each access interval $\mathbf{A}_i$, we will have only one epoch $\mathbf{B}_i$ but in this epoch we move the symbolic addresses not from $\bigcup_{s>\mathbf{a}(i)} L_s$ into $L_{\mathbf{a}(i)}$, but from $\bigcup_{s>\kappa(i)} L_s$ into $L_{\kappa(i)}$, where $\kappa(i)$ is approximately $\mathbf{a}(i) + c_1 \log\log n$. This will have the same effect as the solution with many epochs. We will denote this modified version of the algorithm by $\mathcal{A}$.

In the detailed proof we define $\mathcal{A}$ in several stages. Namely first we look at only the movement of the symbolic adresses on the tree T_n during the algorithm without defining the physical adresses at all. The algorithm which works with the symbolic addresses only will be called $\mathcal{T}$. The advantage of this approach is that the problem of overfilling, which is the main combinatorial difficulty in the proof, can be formulated and solved already in this simplified framework. Naturally the introduction of physical addresses will create a new overfilling problem, since the physical addresses $\mathbf{p}(\tau)$ associated by different nodes τ of the tree can be accidentally the same. Since the choices of the buckets $\mathbf{p}(\tau)$ are completely random and independent, this will not cause a serious problem. *End of sketch.*

Later we will continue the sketch of the proof by giving more detailed information about specific parts.

1 The proof of Lemma 2, basic definitions.

Lemma 2 states that a memory maintenance program P can be simulated by a program P_1 with certain properties. This simulation will be carried out in the following way. Suppose that P has to keep record of the values of n variables $x_0, ..., x_{n-1}$. For the sake of simplicity, and without the loss of generality, we assume that n is a power of 2, namely $n = 2^d$, where d is an integer. We describe an algorithm $\mathcal{A}$ which does the task of P in a $\tilde{\gamma}$-oblivious way. (In the "Sketch" we first described an algorithm $\mathcal{Y}$ doing this and then its modified form, where at the end of epochs we change symbolic addresses so that they will be in level $L_{\kappa(i)}$ and not into level $L_{\mathbf{a}(i)}$, was the algorithm $\mathcal{A}$.) The first $\tilde{\gamma}$ memory cells will be called the protected cells. $\mathcal{A}$ will be implemented by a program running on $\mathcal{M}_q$. This will be the $\tilde{\gamma}$-oblivious simulating program P_1 of Lemma 2. P_1 will be a program of constant length l. One solution would be to make $\tilde{\gamma}$ larger than l and this way the adversary wouldn't get any information about the execution of P_1 other than the memory accesses outside the first $\tilde{\gamma}$ memory cells and the times of these accesses. However it is not necessary to make $\tilde{\gamma}$ as large as P_1. P_1 can be kept outside the first $\tilde{\gamma}$ memory cells and only those part of it brought in which are executed. Since l is a constant, this can be done easily, without revealing which instruction of the program P_1 is executed. Indeed, each time, when a new instruction of P_1 is needed, every memory cell in the block where the program is P_1 is stored will be accessed, but only those values copied into protected cells, which are needed at that moment. The set of memory cells where the program of $\mathcal{A}$ is kept will be denoted by $\mathbf{M}_0$. Apart from that $\mathcal{A}$ will use only the protected cells, that is, the first $\tilde{\gamma}$ memory cells, the memory cells which are in the buckets, and another set poly$(\log n)$ cells. A subset of this last poly$(\log n)$ cells will be the set $\mathbf{X}$ that will be used for temporary storage and that we will call the tray.

Definition. An evaluation pair is a pair $\langle a, v\rangle$ with $a \in \{0,1,...,n-1\}$, $v \in [0,\wp]$. The set of all evaluation pairs will be denoted by E. With each node $\tau \in T_n$ we associate an abstract element $\mathbf{c}(\tau)$ that we will call a coupon, if $\tau \neq \sigma$ then $\mathbf{c}(\tau) \neq \mathbf{c}(\sigma)$. (If we want to represent the coupons by sets, then e.g., $\mathbf{c}(\tau)$ can be the pair $\langle \tau, \emptyset\rangle$.) C will denote the set of all coupons.

Sketch of the proof continued. The motivation for the expression "evaluation-pair" is that when such a pair $\langle a, v\rangle$ will be stored in the memory of $\mathcal{A}$ in a bucket, then the current value of the variable x_a is v. The word coupon is used because of the following reason. Recall that in the Sketch we told that each $\tau \in T_n \backslash \{1_{T_n}\}$ has a symbolic address $\sigma > \tau$. Moreover in the bucket $\mathbf{p}(\sigma)$ a pair of the form $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau)\rangle$ is stored. This pair tells what is the physical address of the bucket assigned to τ. First we intend to give a completely abstract definition of the algorithm, when there are no memory cells or buckets, and we are speaking about everything using only the symbolic addresses. In this case the pair $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau)\rangle$ will be replaced by the coupon $\mathbf{c}(\tau)$, with the meaning that if the algorithm has found the coupon $\mathbf{c}(\tau)$ then it is able to found everything which is stored at the symbolic address τ. In other words, the coupon is a promise that when we translate the algorithm from symbolic addresses to physical addresses, then the coupon $\mathbf{c}(\tau)$ will be translated into the pair $\langle \bar{\tau}, \bar{\mathbf{p}}\tau\rangle$.

We will define the algorithm $\mathcal{A}$ in two steps. First we describe an algorithm $\mathcal{T}$ which does not work on the machine $\mathcal{M}_q$. It places evaluation pairs and coupons at the nodes of the tree T_n and moves around these objects on the nodes of the tree. $\mathcal{T}$ will have inputs and outputs and will handle them in the way that we expect from a memory maintenance program.

Then we define an algorithm $\mathcal{A}$ by a translating the algorithm $\mathcal{T}$ which is manipulating only abstract objects into the algorithm $\mathcal{A}$ which works on $\mathcal{M}_q$. For this translation we will need a (random) function $\mathbf{p}$ which assigns to each node $\tau \in L_i$, $i = 0,1,...,d$ a bucket $\mathbf{p}(\tau) \in \mathcal{B}_i$. Each pair, that was at the node τ in algorithm $\mathcal{T}$, will get into the bucket $\mathbf{p}(\tau)$ in algorithm $\mathcal{D}$. Each coupon $\mathbf{c}(\tau)$ at a node σ in $\mathcal{T}$ will be replaced by a pair $\langle \bar{\tau}, \bar{\mathbf{p}}(\tau)\rangle$ in the bucket $\mathbf{p}(\sigma)$ in algorithm $\mathcal{D}$.

The algorithm $\mathcal{A}$ will work as expected from a memory maintenance program. We also have to make sure also that it is $\tilde{\gamma}$-oblivious.

The advantage of this two step definition of $\mathcal{A}$ instead of defining $\mathcal{A}$ directly, is that all of the critical and new steps of the proof are in the definition of $\mathcal{T}$ and the proof of its basic properties. In particular we will show that $\mathcal{T}$ does not lead to overfilling, in the sense that, with high probability, never will be more than poly$(\log n)$ objects at the same node of the tree. Once we have proved the needed properties of $\mathcal{T}$ the remaining part of the proof, that is making sure that the program $\mathcal{A}$ that we get after the translation is $\tilde{\gamma}$-oblivious, can be done by the methods that has been introduced in [6], [8], [9], and [7]. In particular in this latter part of the proof we will use the following facts/techniques (i) every program which uses only m consecutive memory cells can be made oblivious in deterministic sense, so that the time of the program increases only by a factor of $O(m)$, (ii) the obliviousness of the whole program is proved by noticing that the information that the adversary gets is uniquely determined by the sequence of buckets whose contents the algorithm checks, and we can guarantee that the elements of these bucket sequence are independent and they have predetermined distributions. For this we use only the fact each randomly chosen bucket, in the role for which the randomization has been performed is used, in a way visible to the adversary, at most once. (iii) The random rehashing of the contents of the buckets can be done in an oblivious way using oblivious sorting networks.

Below we define the time intervals $\mathbf{A}_i, \mathbf{B}_i$, $i > 0$ that we will use for the algorithms $\mathcal{T}$ and $\mathcal{A}$ in the roles that we described earlier. We will start with a finer division of the time into intervals I_j, $j \geq 0$. I_0 will be a time interval before all of the intervals $\mathbf{A}_i$, $\mathbf{B}_i$. I_0 will be used for the startup, setting the initial values of the variables, and randomizing χ, and whatever needs to be done a the beginning.

$\mathbf{A}_k$ will consist of $d+3$ subintervals $\mathbf{A}_{k,0}, ..., \mathbf{A}_{i,k+2}$. As we mentioned it earlier the algorithm goes down on the branch $\texttt{branch}(\chi(a_k))$ of the tree where $\langle a_k, v_k, \Psi_k\rangle$ is the kth input that arrives at the beginning of $\mathbf{A}_{k,0}$. The action of the algorithm at level i on this branch will be done in the interval $\mathbf{A}_{k,i}$, for $i = 0, ..., d$. In interval $\mathbf{A}_{d+2}$ the algorithm goes back to the root and algorithm $\mathcal{A}$ puts data that it picked up in the earlier intervals in the bucket $\mathbf{b}_{0,0}$. ($\mathcal{T}$ will perform an analogue task as we will define below.) In the time interval $\mathbf{A}_{k,d+2}$ the algorithm gives the output. *End of sketch.*

The partition of the time into intervals. For the definition of the algorithms $\mathcal{T}$ and $\mathcal{A}$ we partition the time interval $I = [0, \infty)$ into subintervals I_j, $j = 0, 1, ...$ (We will use it with superscripts $I_j^{(\mathcal{T})}$, $I_j^{(\mathcal{A})}$ if we want to make the choice of the algorithm explicit. The various algorithms will perform analogue tasks in these intervals.)

The time interval I_0 will have a special role it is used to set up a data structure used later and also for the choices of certain random values that the algorithm will use later.

For each $k = 0, 1, ...$ the of intervals $I_{(k-1)(d+4)+1}, ..., I_{(k-1)k(d+4)+d+3}$ in this order will be also denoted by $\mathbf{A}_{k,0}, ..., \mathbf{A}_{k,d+2}$ and we define the interval $\mathbf{A}_k$ by $\mathbf{A}_k = \bigcup_{j=0}^{d+2} \mathbf{A}_{k,j}$. The interval $I_{k(d+4)}$ will be denoted by $\mathbf{B}_k$. We will call $\mathbf{A}_k$, $k = 1, 2, 3, ...$ the kth access interval. The algorithm will ask for the input $\langle a_k, v_k, \Psi_k\rangle$, ($a_k \in \{0, 1, ..., n-1\}$, $v_k \in [0, \wp]$, $\Psi_k = 0, 1$) at the beginning of interval $\mathbf{A}_k$, and if $\Psi_k = 0$ stores the value v_k in the memory of $\mathcal{M}$, in some way to be described later. At the end of the time interval $\mathbf{A}_k$, the algorithm provides the output z_k, where $\Psi_k = 0$ implies $z_k = v_k$ and $\Psi_k = 1$ implies that z_k is the current value of variable x_{a_k}.

The time intervals $\mathbf{B}_k$, k=1,2,... will be called epochs or bookkeeping intervals. In the interval $\mathbf{B}_i$ the algorithm will update the data in the buckets in a way that depends on the value of i.

2 The symbolic algorithm $\mathcal{T}$

The algorithm $\mathcal{T}$ as we will define below is "symbolic" in the sense that its memory is not the memory of $\mathcal{M}_q$ but only a set of abstract objects. For the definition of algorithm $\mathcal{T}$ we need a random hash function χ, which assigns to each $i \in \{0, 1, ..., n-1\}$ a $0, 1$ sequence $\chi(i)$ of length d. χ will be randomized with a distribution which has the following property.

Definition. Assume that χ is a random variable whose values are functions defined on $\{0, 1, ..., n-1\}$, and $k \leq n$ is a natural number. The random variable χ is k-wise independent if for each fixed and distinct $i_0, ..., i_{k-1} \in \{0, 1, ..., n-1\}$ the random variables $\chi(i_0), ..., \chi(i_{k-1})$, are mutually independent.

For the algorithm $\mathcal{T}$ we will need a random function χ with $(\log n)^\lambda$-wise independence for some constant $\lambda > 0$, which assigns to each $i \in \{0, 1, ..., n-1\}$ a $0, 1$ sequence $\chi(i)$ of length d. There are well-known methods for generating such a random function, the following one will be suitable for our purposes, since it can be done obliviously and in time poly$(\log n)$.

Generating a random function χ with k-wise independence. χ will be a random polynomial of degree at most $k-1$ over the finite field F_n with $n = 2^d$ elements so that the coefficients of the polynomial are chosen independently and have uniform distribution on F_n. (It may be easier from the point of view of computations in the field if we use a field with p elements, where p is a prime with $n \leq p < 2n$.) k-wise independence follows from the fact that if we prescribe arbitrary values from F_n to the points $i_0, ..., i_{k-1} \in F_n$, then there exists a unique polynomial of degree at most $k-1$ over F_n that takes these values.

Motivation. For the following definition recall that E was defined as the set of all evaluation pairs $\langle a, v \rangle$ and C as the set of all coupons $\mathbf{c}(\tau)$. We define a machine whose each state can be described as a placement of some elements of the set $E \cup C$ by the various nodes of the tree T_n. This will be described by a function F, so that $F(\tau)$, $\tau \in T_n$ is the subset of $E \cup C$ which is at τ. One element of $E \cup C$ can be at most at one node of T_n. Some of the elements which are not at any of the nodes τ of the tree, will be in the set X, which will be called the tray, and will have the role of a temporary storage while the elements of $E \cup C$ are moved from one place on the tree to another.

Definition. We define an algorithm $\mathcal{T} = \mathcal{T}^{(\chi, c_1)}$ which will depend on a function χ defined on $\{0, 1, ..., n-1\}$ whose values are $0, 1$-sequences of length n, and a constant $c_1 > 0$ that we will choose later. For the moment we assume that χ and c_1 are fixed in an arbitrary way. The algorithm $\mathcal{T}$ will work on a machine that we will denote by $\mathcal{R}$ and whose each possible state is a pair $\langle F, X \rangle$, where $X \subseteq E \cup C$ and F is a function F so that the following conditions are satisfied:

(a) $\texttt{domain}(F) = T_n$,

(b) each value of F is a subset of $(E \cup C) \backslash X$,

(c) the various values of F are pairwise disjoint.

When the machine $\mathcal{R}$ is in state $\langle F, X \rangle$, and $\tau \in T_n$, we will say that the elements of $F(\tau)$ are at the node τ of T_n, and the elements of X are in the tray. If a coupon $\mathbf{c}(\tau)$ is at a node of T_n or in the tray then we will say that the coupon is active otherwise it is passive.

We will define the algorithm $\mathcal{T}$ by defining the initial state of the machine $\mathcal{R}$ and then describing how will the algorithm change an arbitrary state of the machine into the next one. The algorithm $\mathcal{T}$ may also ask for inputs and then gets an input of the type $\langle a, v, \Psi\rangle$ that are the inputs of a memory maintenance program, that is $a \in [0, n-1]$, $v \in [0, \wp]$, $\Psi \in \{0, 1\}$.

The initial sate of $\mathcal{T}$ is the pair $\langle F_0, \emptyset\rangle$, where $F_0(\tau) = \emptyset$ for all $\tau \in T_n$.

In the time interval I_0, the algorithm $\mathcal{T}$ does the following:

(5) *$\mathcal{T}$ puts the evaluation-pair $\langle a, 0\rangle$ on $\texttt{leaf}(\chi(a))$ for each $a \in \{0, 1, ..., n-1\}$, and*

(6) *For each $\tau \in T_n \backslash \{1_{T_n}\}$, $\mathcal{T}$ puts the coupon $\mathbf{c}(\tau)$ on τ°, the ancestor node of τ.*

This defines the function F at the end of I_0. Note that the tray X did not change according to this rule during I_0. Therefore we have $X = \emptyset$ at the end of I_0.

As a result of this definition, at the end of I_0, at each non-leaf node σ there are exactly two coupons $\mathbf{c}(\sigma^{(0)})$ and $\mathbf{c}(\sigma^{(1)})$.

We want to define $\mathcal{T}$ in a way that at the beginning of each time interval $\mathbf{A}_i$, $\mathbf{B}_i$, $i > 0$, the state $\langle F, X\rangle$ of $\mathcal{R}$ will satisfy the following conditions:

(7) *$X = \emptyset$*

(8) *for each $a \in \{0, 1, ..., n-1\}$ there exists exactly one $x \in [0, \wp]$ so that $\langle a, x\rangle \in \bigcup_{\tau \in T_n} F(\tau)$. This unique pair $\langle a, x\rangle$ is at a node τ with $\tau \in \texttt{branch}(\chi(a))$.*

(9) *for each $\tau \in T_n \backslash \{1_T\}$, if the coupon $\mathbf{c}(\tau)$ is active then it is at a node σ, with $\sigma > \tau$*

(10) *for all $\tau \in T_n \backslash \{1_{T_n}\}$ statement (a) implies statement (b), where*
(a) $F(\tau) \neq \emptyset$, that is, there exists either at least one evaluation-pair or at least one coupon at node τ,
(b) there exists a $\sigma > \tau$ so that $\mathbf{c}(\tau)$ is at node σ.

Conditions (7), (8) (9), (10) will be called the basic conditions.

Remark. 1. Condition (10) will mean for the algorithm $\mathcal{A}$ that all of the information which are stored in a bucket $\mathbf{p}(\tau)$ can be found by $\mathcal{A}$, if it goes down the branch leading to τ and looks into each bucket $\mathbf{p}(\sigma)$ so that $\sigma \geq \tau$ and $\mathcal{A}$ already knows the address of $\mathbf{p}(\sigma)$. Condition (10) guarantees, that this cumulative process will eventually lead to the address of $\mathbf{p}(\tau)$.

2. In condition (10), statement (b) does not imply statement (a) in general. Suppose for example the two leafs τ_1, τ_2 has a common ancestor σ. If there exist no $a \in \{0, 1, ..., x-1\}$ and $j \in \{1, 2\}$ with $\texttt{leaf}(\chi(a)) = \tau_j$ then after the initial setup at the beginning of interval I_1, the coupons $\mathbf{c}(\tau_j)$, $j = 1, 2$ are at σ but there is nothing at τ_1 and τ_2.

Now we describe what the algorithm $\mathcal{T}$ is doing in each time interval $\mathbf{A}_i$, that is, in an access time interval. At the beginning of the interval the algorithm $\mathcal{T}$ gets the input $\langle a_i, v_i\rangle$. Then $\mathcal{T}$ does the following. In this description the expression $\mathcal{T}$ "destroys a coupon/pair" means that the coupon/pair will be neither at a node of T_n nor in the set X. (The coupon becomes passive.)

As we said earlier the time interval $\mathbf{A}_i^{(\mathcal{T})}$ is partitioned into $d+3$ subintervals $\mathbf{A}_{i,0},...,\mathbf{A}_{i,d+1},\mathbf{A}_{i,d+2}$. We describe the behavior of $\mathcal{T}$ in each of these intervals separately. We assume that at the beginning of interval $\mathbf{A}_{i,0}$ the basic conditions are satisfied.

The definition $\mathcal{T}$ in interval $\mathbf{A}_{i,0}$.

(11) *$\mathcal{T}$ puts all the coupons and evaluation pairs that are at node 1_{T_n} into the tray.*

The definition of $\mathcal{T}$ in the interval $\mathbf{A}_{i,s}$, for $s=1,2,...,d$.

(12) *Assume that $L_s^{(T_n)} \cap \texttt{branch}(\chi(a_i)) = \{\tau\}$.*

Case I. The coupon $\mathbf{c}(\tau)$ is not in the tray at the beginning of $\mathbf{A}_{i,s}$. *In this case $\mathcal{T}$ does not do anything.*

Case II. The coupon $\mathbf{c}(\tau)$ is in the tray at the beginning of $\mathbf{A}_{i,s}$. *In this case $\mathcal{T}$ puts all of the coupons and evaluation pairs that are at node τ into the tray, and destroys the coupon $\mathbf{c}(\tau)$.*

The definition $\mathcal{T}$ in interval $\mathbf{A}_{i,d+1}$.

(13) *$\mathcal{T}$ puts down to the node 1_{T_n} all of the coupons and evaluation-pairs that are in the tray. (The tray becomes empty.)*

We will show that it is a consequence of the basic conditions, that after step (13) there is always a pair of the form $\langle a_i, x\rangle$ at node 1_T. Therefore, $\mathcal{T}$ is always able to execute the following step.

The definition $\mathcal{T}$ in interval $\mathbf{A}_{i,d+2}$.

(14) *assume that $\langle a_i, x\rangle$ is the unique evaluation pair at node 1_{T_n} whose first element is a_i. If the input at the beginning of the interval $\mathbf{A}_i$ was $\langle a_i, v_i, \Psi_i\rangle$ with $\Psi_i = 1$, then $\mathcal{T}$ gives the output x. If $\Psi_i = 0$ then $\mathcal{T}$ destroys the pair $\langle a_i, x\rangle$, creates a pair $\langle a_i, v_i\rangle$, places it at node 1_{T_n}, and gives the output v_i.*

Now we describe what the algorithm $\mathcal{T}$ is doing in the epochs $\mathbf{B}_i$, $i=1,2,....$

The definition of $\mathcal{T}$ in the time interval $\mathbf{B}_i^{(\mathcal{T})}$. Let $\ell = \ell(i)$ be the largest power of two which is a divisor of i, and let ν be the smallest positive integer with $2^\nu > d^{c_1}$. For all $i=1,2,...$ we define an positive integer κ, depending on i, by $\kappa = \kappa(i) = \min\{d, 1+\nu+\log_2 \ell(i)\}$. $\mathcal{T}$ executes the following steps, in the given order, in the time interval $\mathbf{B}_i^{(\mathcal{T})}$,

(15) *$\mathcal{T}$ destroys all of the coupons $\mathbf{c}(\tau)$, where $\tau \in \mathbf{L}_\kappa = \bigcup_{j\le\kappa} L_j$.*

(16) *$\mathcal{T}$ for each $\rho \in \mathbf{L}_{\kappa-1}$, if a coupon $\mathbf{c}(\tau)$, is at the node ρ, then $\mathcal{T}$ puts $\mathbf{c}(\tau)$ at the unique node σ in L_κ with $\sigma \ge \tau$. (Such a node exists, otherwise $\mathbf{c}(\tau)$ would have been destroyed in step* (15)*.)*

(17) *For all $\tau \in \mathbf{L}_{\kappa-1}$ and for all evaluation pairs $\langle a, v\rangle$ that are at node τ, $\mathcal{T}$ puts $\langle a, v\rangle$ to the unique node of L_κ which is also in* $\texttt{branch}(\chi(a))$*.*

(18) *For each $\tau \in \mathbf{L}_\kappa \in \backslash\{1_{T_n}\}$, $\mathcal{T}$ activates the coupon $\mathbf{c}(\tau)$ and puts it on τ°.*

This completes the definition of the algorithm $\mathcal{T}$. The definition of $\mathcal{T}$ depended on the choice of a function χ defined on $\{0,1,...,n-1\}$ and with values in $\{0,1\}^d$, and on the choice of a constant $c_1 > 0$, If we want to make this dependence explicit we will write $\mathcal{T}^{(\chi,c_1)}$ instead of $\mathcal{T}$.

Lemma 3 *For all possible choices of the function χ and the constant $c_1 > 0$, if $\mathcal{T} = \mathcal{T}^{(\chi,c_1)}$, then the following holds. The basic conditions are satisfied at the end of time interval I_0 of algorithm $\mathcal{T}$. Moreover, for each $i \geq 1$, if the basic conditions are satisfied at the beginning of a time interval $\mathbf{A}_i$ of $\mathcal{T}$, they will be also satisfied at the end of the interval $\mathbf{A}_i$, and the same is true for all of the time intervals $\mathbf{B}_i$, $i \geq 1$. Consequently the algorithm $\mathcal{T}$ is well-defined and the basic conditions remain true at the end of each time interval $\mathbf{A}_i, \mathbf{B}_i$, $i \geq 1$ while $\mathcal{T}$ is working.*

Proof. First we show that the basic conditions are satisfied at the end of time interval I_0.

Condition (7) As we have noted already, at the initial state of the machine $X = \emptyset$, and nothing is put in the tray during I_0.

Condition (8). In step (5) for each $a \in \{0,1,...,n-1\}$ exactly one evaluation pair of the form $\langle a, x\rangle$, namely $\langle a, 0\rangle$ is put down at a node of T_n, and in the remaining step (6) no evaluation pairs are added or removed.

Condition (9). This is a consequence of the fact that all of the coupons which became active in interval I_0, were added in step (6), where coupon $\mathbf{c}(\tau)$ is placed at node $\tau^\circ > \tau$.

Condition (10). For interval I_0 this is a consequence of condition (9), since in step (6) the algorithm $\mathcal{T}$ makes all of the coupons $\mathbf{c}(\tau)$ active for $\tau \in T_n\backslash\{1_{T_n}\}$.

As a next step we show that the assumption that we have used in the definition $\mathbf{A}_{i,d+2}$ (step (14)) was justified. We have to show that

Proposition 1 *Assume that the basic conditions are satisfied at the beginning of interval $\mathbf{A}_i$. The definition of the action of $\mathcal{T}$ in interval $\mathbf{A}_{i,d+2}$ is well defined, that is after step* (13) *there is always a pair of the form $\langle a_i, x\rangle$ at node 1_{T_n}.*

To prove this first we prove the following fact.

Proposition 2 *Assume that at the beginning of the time interval $\mathbf{A}_i$, $i \geq 1$, the basic conditions are satisfied. Suppose further that $w \in E \cup C$ and w is at a node $\tau \in \mathtt{branch}(\chi(a_i))$ at the beginning of interval $\mathbf{A}_i$. If $\tau \in L_s$, $s \in \{0,1,...,d\}$, then $\mathcal{T}$ puts the object w in the tray in the time interval $\mathbf{A}_{i,s}$.*

Proof. We prove the proposition by induction on s. For $s = 0$ the statement of the proposition is an immediate consequence of the definition of step (11) in the description of $\mathcal{T}$. Assume that the statement of the proposition holds if $\tau \in \bigcup_{r<s} L_r$. Suppose that $w \in E \cup C$ and w is at the node τ with $\tau \in L_s$. Condition (10) of the "basic conditions" imply that at the beginning of $\mathbf{A}_i$ the coupon $\mathbf{c}(\tau)$ is at a node $\sigma > \tau$, where $\sigma \in L_r$, $r < s$. Therefore by the inductive assumption the coupon $\mathbf{c}(\tau)$ was placed in the tray by $\mathcal{T}$ in the time interval $\mathbf{A}_{i,r}$. The definition of step (12), that is, the action of $\mathcal{T}$ in the intervals $\mathbf{A}_{i,j}$ for $j = r, ..., s-1$, implies that at the beginning of $\mathbf{A}_{i,s}$ the coupon $\mathbf{c}(\tau)$ is still in the tray. Therefore by Case II of step (12) w is put in the tray in interval $\mathbf{A}_{i,s}$ as claimed. *Q.E.D.*(Proposition 2)

We can prove now proposition 1. At the beginning of $\mathbf{A}_i$ the basic conditions are satisfied so by condition (8) there exists a $\tau \in \mathtt{branch}(\chi(a))$ and an $x \in [0, \wp]$ so that $\langle a_i, x\rangle$ is at τ. Therefore according to Proposition 2, in some of the intervals $\mathbf{A}_{i,0}, ..., \mathbf{A}_{i,d}$ the pair $\langle a_i, x\rangle$ is put in the tray and according to the definition of step (12) it remains there till the beginning of interval $\mathbf{A}_{i,d+1}$. In $\mathbf{A}_{i,d+1}$ the pair $\langle a_i, x\rangle$ is put down at the node 1_{T_n} as claimed. *Q.E.D.*(Proposition 1)

Now we show that at the end of $\mathbf{A}_j$ the state of the machine $\mathcal{R}$, satisfies the basic conditions.

Condition (7) At the end of interval $\mathbf{A}_{i,d+1}$ the tray is empty as it is described in step (13). In the interval $\mathbf{A}_{i,d+2}$ nothing is put into the tray according to the definition of step (14).

Condition (8). During the interval $\mathbf{A}_i$ no element $\langle a, v\rangle \in E$ was destroyed or created with $a \neq a_i$. Moreover all pairs in E which were moved during this interval were moved to the node 1_{T_n} which is contained in all of the sets $\mathtt{branch}(\chi(a_i))$. Therefore, for each $a \neq a_i$, if condition (8) was satisfied at the beginning of $\mathbf{A}_i$ it will be also satisfied at the end of this time interval. Consequently we only have to check the statement for $a = a_i$. Since at the beginning of $\mathbf{A}_i$ the basic conditions were satisfied, there was a unique pair of the form $\langle a_i, x\rangle$ in $\bigcup F_{\tau\in T_n}(\tau)$ at the beginning of $\mathbf{A}_{i,d+2}$ since till then no element of E was created or destroyed. According to the Proposition 1 at the beginning of interval $\mathbf{A}_{i,d+2}$ there is a pair $\langle a_i, x\rangle$ at node 1_{T_n}. The definition of step (14), implies that at the end of $\mathbf{A}_{i,d+2}$ there is a unique pair of the form $\langle a_i, y\rangle$ at 1_{T_n} but no pair of this form at any other nodes. Since $1_{T_n} \in \mathtt{branch}(\chi(a_i))$ this implies that condition (8) is satisfied.

Condition (9). Assume that $\tau \in T_n\backslash\{1_{T_n}\}$ and the coupon $\mathbf{c}(\tau)$ is active at the end of $\mathbf{A}_i$. If the coupon $\mathbf{c}(\tau)$ was not moved during interval $\mathbf{A}_i$ then the condition is satisfied because it was satisfied at the beginning of $\mathbf{A}_i$. If the coupon was moved during $\mathbf{A}_i$ then it is at 1_{T_n}, since the only time when coupons were placed at nodes, during the time interval $\mathbf{A}_i$, was in interval $\mathbf{A}_{i,d+1}$, in step (13). In the next interval, in interval $\mathbf{A}_{i,d+2}$, no coupon was moved. Therefore $\sigma = 1_{T_n}$. Since $\tau \neq 1_{T_n}$ we have $\sigma > \tau$ as claimed.

Condition (10). Suppose that $\tau \neq 1_{T_n}$ and at the end of $\mathbf{A}_i$ statement (a) of condition (10) holds. We claim that $\tau \notin \mathtt{branch}(\chi(a_i))$. Indeed assume $\tau \in \mathtt{branch}(\chi(a_i))$. By Proposition 2 all of the objects form node τ was put into the tray in interval $\mathbf{A}_{i,s}$ where $\tau \in L_s$. The only node where objects are placed during $\mathbf{A}_i$ is 1_{T_n} and $\tau \neq 1_{T_n}$. We reached a contradiction so $\tau \notin \mathtt{branch}(\chi(a_i))$.

Since the basic condition were satisfied at the beginning of interval $\mathbf{A}_i$ at that time the coupon $\mathbf{c}(\tau)$ was at a node $\sigma > \tau$. If $\sigma \notin (\mathtt{branch}(\chi(a_i)))$ then nothing was put in the tray from the objects that are at σ during $\mathbf{A}_i$ so statement (b) of condition (10) holds with the same σ at the end of $\mathbf{A}_i$. Assume now that σ is in $\mathtt{branch}(\chi(a_i))$. Then by Proposition 2 coupon $\mathbf{c}(\tau)$ is put in the tray by $\mathcal{T}$. Some of the coupons are destroyed during the interval $\mathbf{A}_{i,s}$ as described in Case II of step (12). However according to this definition a coupon $\mathbf{c}(\tau)$ can be destroyed only if $\tau \in \mathtt{branch}(\chi(a_i))$. Since we know that $\tau \notin \mathtt{branch}(\chi(a_i))$, $\mathbf{c}(\tau)$ is not destroyed, and so in interval $\mathbf{A}_{i,d+1}$ is placed at the node 1_{T_n} where it remains in interval $\mathbf{A}_{i,d+2}$. By the assumption of condition (10) $\tau \neq 1_{T_n}$, consequently statement (b) of that condition holds at the end of time interval $\mathbf{A}_i$ with $\sigma = 1_{T_n}$.

This completes the proof of the fact that the basic conditions are satisfied at the end of time interval $\mathbf{A}_i$, provided that they were satisfied at the beginning of this time interval.

Assume that for some $i = 1, 2, ...$ the basic conditions are satisfied at the beginning of the time interval $\mathbf{B}_i$. We show that they are also satisfied at the end of this time interval.

Condition (7) trivially holds because the tray was not used during the interval $\mathbf{B}_i$

Condition (8). Assume that an $a \in \{0, 1, ..., n-1\}$ is fixed. During the time interval $\mathbf{B}_i$, no evaluation pairs are destroyed or created so the condition that there exists exactly one $x \in [0, \wp]$ with $\langle a, x\rangle \in \bigcup_{\tau \in T_n} F(\tau)$ trivially remains true at the end of $\mathbf{B}_i$. Evaluation pairs are moved only in step (17) during $\mathbf{B}_i$. In this step if a pair $\langle a, v\rangle$ is moved at all, then it is placed at a node in $\mathtt{branch}(\chi(a))$ and remains there till the end of $\mathbf{B}_i$ as claimed.

Condition (9). Assume that $\tau \in T_n \backslash \{1_{T_n}\}$ and the coupon $\mathbf{c}(\tau)$ is active at the end of interval $\mathbf{B}_i$. We use the notation introduced in the definition of $\mathcal{T}$ in $\mathbf{B}_i$. If $\tau \in \mathbf{L}_\kappa$ then steps (15) and (18) imply that the coupon $\mathbf{c}(\tau)$ is on node $\tau^\circ > \tau$ satisfying condition (9).

Assume now that $\tau \notin \mathbf{L}_\kappa$ and $\mathbf{c}(\tau)$ is active at the end of $\mathbf{B}_i$. The only step during $\mathbf{B}_i$ where coupons are activated is step (18). The coupon $\mathbf{c}(\tau)$, however was not activated here because $\tau \notin \mathbf{L}_\kappa$. Therefore $\mathbf{c}(\kappa)$ was already active at the beginning of $\mathbf{B}_i$. Assume that at that time it was at node $\eta \in T_n$. Since (9) was satisfied at the beginning of $\mathbf{B}_i$, we have that $\eta > \tau$. Coupons are moved only in step (16) during $\mathbf{B}_i$. By the definition of this step if $\eta \notin \mathbf{L}_{\kappa-1}$ then $\mathbf{c}(\tau)$ is not moved during $\mathbf{B}_i$ so at the end of $\mathbf{B}_i$ condition (9) is satisfied with $\sigma = \eta$. Assume now that $\eta \in \mathbf{L}_{k-1}$. Then by step (16) $\mathbf{c}(\tau)$ is moved to a node $\sigma \in L_\kappa$ with $\sigma \geq \tau$. By our assumption however $\tau \notin \mathbf{L}_\kappa \supseteq L_\kappa$ therefore $\sigma \neq \tau$, and so $\sigma > \tau$ as it is required.

Condition (10). Assume that $\tau \in T_n \backslash \{1_{T_n}\}$ and there is at least one element of the set $E \cup C$ at τ. Suppose first that $\tau \in \mathbf{L}(\kappa)$. In this case in step (18) the coupon $\mathbf{c}$ is placed at the node τ° so statement (b) holds with $\sigma = \tau^\circ$. Assume now that $\tau \notin \mathbf{L}_\kappa$. Then by the definition of $\mathcal{T}$ in $\mathbf{B}_i$, nothing was placed at τ during $\mathbf{B}_i$, therefore statement (a) of condition (10) was satisfied at the beginning of $\mathbf{B}_i$ as well. Since the basic conditions were all satisfied at that time, we get that statement (b) was true at the beginning of $\mathbf{B}_i$, and so $\mathbf{c}(\tau)$ was at a node σ with $\sigma > \tau$. If $\sigma \notin \mathbf{L}_{\kappa-1}$ then $\mathbf{c}(\tau)$ is still at σ at the end of $\mathbf{B}_i$. Assume now that $\sigma \in \mathbf{L}_{\kappa-1}$. Then $\mathbf{c}(\tau)$ is moved in step (16) to an element of $\sigma' \in L_\kappa$ with $\sigma' \geq \tau$ and $\mathbf{c}(\tau)$ remains there till the end of $\mathbf{B}_i$. Since $\tau \notin \mathbf{L}_\kappa \supseteq L_\kappa$ we have $\tau \neq \sigma'$ and so $\tau < \sigma'$. Consequently at the end of $\mathbf{B}_i$ statement (b) is true with σ' in the role of σ. *Q.E.D.*(Lemma 3)

Lemma 4 *For all possible choices of the function χ and the constant $c_1 > 0$, if $\mathcal{T} = \mathcal{T}^{(\chi, c_1)}$ then the following holds. input sequence $\langle a_1, v_1, \Psi_1\rangle, ..., \langle a_m, v_m, \Psi_m\rangle$, with $a_i \in \{0, 1, ..., n-1\}$, $v_i \in [0, \wp]$, $\Psi_i \in \{0, 1\}$ the program $\mathcal{T}$ gives the output z_i in the interval $\mathbf{A}_i$ what is expected from a memory maintenance program after the input $\langle a_i, v_i, \Psi_i\rangle$. More precisely the following holds. If $\Psi_i = 0$ then the $z_i = v_i$. If $\Psi_i = 1$ and there exists a largest positive integer j with “$j < i \wedge \Psi_j = 1$ then $z_i = v_j$. Otherwise $z_i = 0$.*

Proof. The statement of the lemma is an immediate consequence of Proposition 1 and the definition of $\mathcal{T}$ in the interval $\mathbf{A}_{i,d+2}$ as described in step (14). *Q.E.D.*(Lemma 4)

Remark. The previous two lemmas were valid for each fixed choice of χ. The k-wise independence property of χ will be used in the next lemma which says that with high probability the total number of items, coupons and evaluation pairs, remain below a poly$(\log n)$ bound during the execution of the tree algorithm $\mathcal{T}$, if the number of inputs is only polynomial in n.

Definition. The set of all 0, 1-sequences of length i will be denoted by $\{0,1\}^i$.

Lemma 5 *There exists $c_1 > 0, \gamma > 0$ $\lambda > 0$ so that for all sufficiently large integers n and for all input sequences $\langle a_i, v_i, \Psi_i \rangle$, $i = 1, ..., j$ of length at most $n^{\log n}$, if the values of the random variable χ are functions defined on $\{0, 1, ..., n-1\}$ with values in $\{0,1\}^n$ and χ is $(\log n)^{\lambda}$-wise independent, then with a probability of at least $1 - n^{-\log n}$ the following holds. For all $\tau \in T_n$ the number of evaluation pairs and coupons at the node τ, remains below $(\log n)^{\gamma}$ during the execution of algorithm $\mathcal{T}^{(\chi, c_1)}$.*

3 Proof of Lemma 5

Notation. Recall the following notation introduced in the definition of $\mathcal{T}^{(\chi,c_1)}$ at the definition of the action of $\mathcal{T}$ in the epoch $\mathbf{B}_i$. $\ell = \ell(i)$ is the largest power of two which is a divisor of i, ν is the smallest positive integer with $2^{\nu} > d^{c_1}$, and for all $i = 1, 2, ..., \kappa = \kappa(i) = \min\{d, 1+\nu+\log_2 \ell(i)\}$.

Definition. Each natural number m will be considered as the set of all natural numbers less than m. That is $0 = \emptyset$, $1 = \{\emptyset\}$, $2 = \{\emptyset, \{\emptyset\}\}$, ..., $m = \{0, 1, ..., m-1\}$.

Proof of Lemma 5. We reduce the lemma to special cases where the computation of probabilities is easier.

Proposition 3 *Assume that $c_1 \geq 2$.*

(a) if Lemma 5 holds with the additional assumption $\tau \neq 1_{T_n}$, then it holds in its original form as well.

(b) if Lemma 5 holds with the additional assumption $\tau \notin \mathbf{L}_{\nu}$, then it holds in its original form as well.

Proof of proposition 3. First we prove the weaker statement (a) of the proposition. Consider the first time t when the number of evaluation-pairs at node 1_{T_n} is greater than M, where $M > 2$ is an arbitrary but fixed number. Clearly this cannot be in a bookkeeping interval $\mathbf{B}_i$ since at the end of such an interval the number of objects at 1_{T_n} is 0, namely the coupons of the successors of 1_{T_n} are there and nothing else.

Therefore t must be in an access interval $\mathbf{A}_i$. Each interval $\mathbf{A}_i$ comes right after an interval $\mathbf{B}_{i-1}$ or the interval I_0. In either case there are exactly two coupons at 1_{T_n} at the beginning of $\mathbf{A}_i$ and no other objects. Consequently if during $\mathbf{A}_i$ the number of objects at 1_{T_n} reaches M, then at least $M' = M - 2$ of them is placed there during $\mathbf{A}_i$.

Throughout interval $\mathbf{A}_i$, the only occasion when algorithm $\mathcal{T}$ puts objects at the node 1_{T_n} is in interval $\mathbf{A}_{i,d+1}$, when $\mathcal{T}$ puts the objects from the tray to node 1_{T_n}. By the definition of $\mathcal{T}$, $\mathcal{T}$ puts an object q in the tray during $\mathbf{A}_i$ only if q was at a node in the branch $B = \mathtt{branch}(\chi(a_i))$ at the beginning of $\mathbf{A}_i$. Therefore the total number of objects at the d nodes of $B \backslash 1_{T_n}$ at the beginning of $\mathbf{A}_i$ was at least M' and therefore there was a node in B with at least M'/d objects at the beginning of algorithm $\mathcal{T}$. Consequently if we prove that the number of objects at each node with the exception of 1_{T_n} is at most d^{γ}, then without the exception we will have the upper bound $2 + d^{\gamma+1} \leq d^{\gamma+2}$ if $d \geq 2$, $\gamma \geq 1$. This completes the proof of statement (a) of the proposition.

Throughout interval $\mathbf{A}_i$, the only occasion when algorithm $\mathcal{T}$ puts objects to a node of $\mathcal{T}$ is in interval $\mathbf{A}_{i,d+1}$, when $\mathcal{T}$ puts the objects from the tray to node 1_{T_n}.

In a bookkeeping interval new objects appear at the nodes in the following way. At the end of a bookkeeping interval $\mathbf{B}_i$, at each node $\tau \in \mathbf{L}_{\kappa(i)-1}$ there are exactly two objects at τ, namely the coupons $\mathbf{c}(\tau^{(0)})$ and $\mathbf{c}(\tau^{(1)})$, where $\tau^{(0)}$ and $\tau^{(1)}$ are the successors of τ. The definition of $\kappa(i)$ implies that $\mathbf{L}_\nu \subseteq \mathbf{L}_{\kappa(i)-1}$ for all $i = 1, 2, ...$, therefore at the end of a bookkeeping interval the number of objects at each $\tau \in \mathbf{L}_\nu$ is at most 2.

These two observations imply that if the number of objects are greater than 2 at a node $\tau \in \mathbf{L}_\nu$ then $\tau = 1_{T_n}$. This together with the already proven statement (a) implies statement (b). *Q.E.D.*(Proposition 3)

The choice of λ. We estimate the number of objects at the nodes of $\mathbf{L}_d$. The d^λ-wise independence implies that for each fixed $b \in L_d$ the probability p_b that there are at least k integers $a \in n$ with $\mathtt{leaf}(\chi(a)) = b$ is at most $\binom{n}{k} n^{-k} \leq (k!)^{-1}$. For $\lambda \geq 2$ and $k = d^2$ we get $k! = ((\log n)^2)! \geq (\log n)^{\frac{1}{2}(\log n)^2} \geq n^{\frac{1}{2}\log n \log\log n}$ which implies that $p_b \geq n^{-\frac{1}{2}\log n \log\log n}$. Therefore

Proposition 4 *with* $\lambda \geq 2$ *in the choice of* χ, *the probability that*

(19) *there exists a leaf* $b \in L_d$, *so that that the number of integers* $|\{a \in n \mid \mathtt{leaf}(\chi(a)) = b\}| \geq (\log n)^2$

is at most $n^{-\frac{1}{2}\log n \log\log n} \geq n^{-4\log n}$.

At this point we do not fix yet λ, but we assume that it will be at least 2 and so, with high probability for the randomization of χ, condition (19) will not hold. In Lemma 5 we are speaking about the number of all objects evaluation pairs and coupons at a node τ of T_n the following proposition shows that for a poly$(\log n)$ bound on this number it is sufficient to count the evaluation pairs.

Proposition 5 *During the execution of* $\mathcal{T}$, *at each time* t, *and for each* $\tau \in T_n$, $N_e(\tau, t)$ *denotes the number of evaluation pairs at* τ, *and* $N_c(\tau, t)$ *denotes the number of coupons at* τ. *Let* $M = \max_{\tau,t} N_e(\tau, t)$. *Then* $\max_{t,\tau} N_c(\tau, t) \leq 1 + 2M \log n$. *Consequently, it is sufficient to prove Lemma 5, in the modified form, where its conclusion is the following. "For all* $\tau \in T_n \backslash \mathbf{L}_\nu$, *the number of evaluation-pairs at* τ *remains below* $(\log n)^\gamma$ *during the execution of algorithm* $\mathcal{T}$*."*

Proof of Proposition 5 We will call a coupon which has not been moved yet after its last activation, a stationary coupon. We will need the following about stationary coupons.

Claim 1 *for all* $\tau \in T_n$, *while* $\mathcal{T}$ *is running, there can be at most* 2 *stationary coupons at* τ.

Proof of Claim 1. The statement of the claim is a consequence of the fact that that coupons are activated only in the epochs $\mathbf{B}_i$ and at the time when a coupon $\mathbf{c}(\tau)$ is activated it is put down at the node τ° together with the coupon $\mathbf{c}(\tau')$, where τ' is the other successor of the node τ°. Moreover in this case we have $\tau \in \mathbf{L}_{\kappa(i)}$, and so $\tau^\circ \in \mathbf{L}_{\kappa(i)-1}$, and therefore all of the coupons which previously were at τ° were destroyed (deactivated) earlier in $\mathbf{B}_i$. This guarantees that each coupon becomes active together with only a single other coupons at the same node, and until it is destroyed other coupons are activated at the same node. This completes the proof of Claim 1

For each $a \in \{0, 1, ..., n-1\}$ at each time while $\mathcal{T}$ is working we will denote by $\mathtt{pair}(a)$ the unique evaluation pair of the form $\langle a, v\rangle$ so that there exists a $\tau \in T_n$ that $\langle a, v\rangle$ is at τ

Claim 2 *Assume that $a \in \{0, 1, ..., n-1\}$, $a \in T_n$ and the coupon $\mathbf{c}(\tau)$ is a stationary coupon at the beginning of $\mathbf{A}_i$, an during $\mathbf{A}_i$ both $\mathbf{c}(\tau)$ and $\mathtt{pair}(a)$, where in $\mathbf{A}_i$ the input was $\langle a_i, v_i, \Psi\rangle$ with $a_i = a$. Then $\tau^{\circ} \in \mathtt{branch}(\chi(a))$ and starting with the end of A_i till the time the coupon $\mathbf{c}$ is first destroyed, the evaluation-pair $\mathtt{pair}(a)$ and the $\mathbf{c}(\tau)$ are always at the same node.*

Proof of Claim 2. The definition of $\mathcal{T}$ in $\mathbf{A}_i$ implies that both $\mathtt{pair}(a)$ and $\mathbf{c}(\tau)$ are put down from the tray to the node 1_{T_n} at the end of $\mathbf{A}_i$.

Assume now that $\mathbf{c}(\sigma)$ and $\mathtt{pair}(a)$ are together at the same node at the beginning of an access interval $\mathbf{A}_i$. If any of them is put into a the tray then the other one will be put there as well, so at the end of the interval they will be again together at node 1_{T_n} (unless $\mathbf{c}(\tau)$ is destroyed). If they are not put into the tray the naturally the will remain at the same node at the end of $\mathbf{A}_i$.

During an interval $\mathbf{B}_i$ the following happens. The fact that $\mathbf{c}(\tau)$ was a stationary coupon at the beginning of $\mathbf{A}_i$ implies that it was at node τ° at that time and so $\tau^{\circ} \in \mathtt{branch}(\chi(a))$. Therefore the path between 1_{T_n} and τ° is contained in $\mathtt{branch}(\chi(a))$. Consequently in $\mathbf{B}_i$ both $\mathtt{pair}(a)$ and $\mathbf{c}(\tau)$ are moved to the unique element of $L_{\kappa(i)} \cap \mathtt{branch}(\chi(a))$ unless $\mathbf{c}(\tau)$ is destroyed at the beginning of $\mathbf{B}_i$. *Q.E.D.*(Claim 2)

Assume that $\sigma, \tau \in T_n$ and at time t, $\mathbf{c}(\tau)$ is a non-stationary coupon at σ. Since $\mathbf{c}(\tau)$ is not stationary it was put in the tray at some time, say, this happened first in interval $\mathbf{A}_i$ where the input was $\langle a_i, v_i, \Psi_i\rangle$. By Claim 2 $\mathtt{pair}(a_i)$ is also at σ at time t. The integer a_i that we defined this way depends in t, σ and τ, so we will denote it by $a(t, \sigma, \tau)$. By Claim 2, we have $\tau^{\circ} \in \mathtt{branch}(\chi(a(t, \sigma, \tau)))$. Therefore for fixed t and σ, the number of different elements $\tau \in T_n$ so that $a(t, \sigma, \tau)$ is the same can be at most $2 \log n$. Therefore the number of coupons at time t at σ is at most the number of stationary coupons there plus $2 \log n$ times the number of evaluation pairs at σ at the same time. Claim 1 gives the upper bound 2 on the stationary coupons, so we get the upper bound $2 + 2 \log nM$ on the total number of coupons at time t at σ. *Q.E.D.*(Proposition 5)

Assume that k_0, M_0 are positive integers, χ has k_0-wise independence, $u \in T_n \backslash \{\mathbf{L}_{\nu}\}$. Let t be a fixed time while the algorithm $\mathcal{T}$ is running. We estimate the probability (for fixed χ and a fixed input, with respect to the randomizations in the bookkeeping intervals $\mathbf{B}_i$) that it happens first at time t, that the number of objects at u is greater than M_0. The time t cannot be in an access interval $\mathbf{A}_i$, since in such an interval objects placed only to 1_{T_n}. Therefore it may be only in I_0 or in a bookkeeping interval $\mathbf{B}_i$. Proposition 4 implies that if $k_0 \geq d^{\lambda}, M_0 \geq d^{\gamma}$ where $\lambda > 0, \gamma > 0$ are sufficiently large constants then t cannot be in I_0. Suppose that t is in the interval $\mathbf{B}_i$. We assume now that at the end of an interval $\mathbf{B}_i$ the number of objects at node u is greater than M_0.

The only level where the number of objects increases in interval $\mathbf{B}_i$ is $L_{\kappa(i)}$. Recall that

$\kappa(i) = \min\{d, 1 + \nu + \log_2 \ell(i)\}$. For later use we note that it is a consequence of $u \notin \mathbf{L}_\nu$ and the definition of ν as the smallest integer with $2^\nu > d^{c_1}$, that for all sufficiently large n we have

(20) $\nu < \kappa(i)$, *and* $d^{c_1} < 2^{\kappa(i)-1}$

Since the number of evaluation pairs at u becomes greater than M_0 the first time during the execution of $\mathcal{T}$ in the interval $\mathbf{B}_i$, we have $u \in L_{\kappa(i)}$, otherwise the number of pairs couldn't have grown at u in $\mathbf{B}_i$. We also know by Proposition 4 that with high probability for the randomization of χ that $\kappa(i) \neq d$. We assume that such a function χ is fixed. Let $\mu = \nu + 1$. We write $i2^\mu$ in the form $i2^\mu = \alpha n + i'2^\mu i$, where α, i' are natural numbers and $i'2^\mu < n$. Let $i'2^\mu = \sum_{j=0}^{r-1} 2^{\psi_j}$, where $\kappa(i) = \psi_0 < \psi_1 < ... < \psi_{r-1} < d$. $\kappa(i) = \psi_0$ is a consequence of the definition of $\kappa(i)$ and $\psi_0 < d$. We define ψ_r by $\psi_r = d$.

Let $\beta_0 = i$ and let $2^\mu \beta_j = i2^\mu - (2^{\psi_0} + ... + 2^{\psi_{j-1}})$ for $j = 1, ..., r$. We may also write this in the form

(21) $\beta_j = 2^{-\mu}\alpha + 2^{\psi_{r-1}-\mu} + ... + 2^{\psi_{j+1}-\mu} + 2^{\psi_j-\mu}$

Clearly $\beta_r = \alpha n 2^{-\mu}$. The definition of $\kappa(\beta_j)$ and $\psi(j) \leq d$ for $j = 0, 1, ..., r-1$ implies $\kappa(\beta_j) = \psi_j$ for $j = 0, 1, ..., r-1$. We consider the bookkeeping intervals $\mathbf{B}_{n/2^\mu} = \mathbf{B}_{\beta_r}, \mathbf{B}_{\beta_{r-1}} ..., \mathbf{B}_{\beta_0} = \mathbf{B}_i$. We claim that

Proposition 6 *The number of access intervals $\mathbf{A}_l$ between $\mathbf{B}_{\beta_{j+1}}$ and $\mathbf{B}_{\beta_j}$ is at most $\beta_j - \beta_{j+1} = 2^{\psi_j - \mu} \leq d^{-c_1} 2^{\psi_j}$.*

Proof. The statement that the number of access intervals $\mathbf{A}_l$ between $\mathbf{B}_{\beta_{j+1}}$ and $\mathbf{B}_{\beta_j}$ is at most $\beta_j - \beta_{j+1}$ is an immediate consequence of the fact that access and bookkeeping intervals alternate. *Q.E.D.*(Proposition 6)

Proposition 7 *Assume that $\beta_{j+1} < \zeta < \beta_j$. Then $\kappa(\zeta) < \kappa(\beta_j) = \psi_j < \kappa(\beta_{j+1}) = \psi_{j+1}$.*

Proof. By (21) we have $\beta_j = 2^{-\mu}\alpha + 2^{\psi_{r-1}-\mu} + ... + 2^{\psi_{j+1}-\mu} + 2^{\psi_j-\mu}$ and $\beta_j = 2^{-\mu}\alpha + 2^{\psi_{r-1}-\mu} + ... + 2^{\psi_{j+1}-\mu}$

The inequality $\beta_{j+1} < \zeta < \beta_j$ implies that if 2^z is the largest power of 2 which divides $2^\mu\zeta$ then 2^z is smaller than 2^{ψ_j}. Since $\psi_j < d$ this implies $\kappa(\zeta) = z$. Together with $\kappa_{\beta_j} = \psi_j$ and $\kappa_{\beta_{j+1}} = \psi_{j+1}$ this implies the statement of the proposition. *Q.E.D.*(Proposition 7)

Let A_j be the set of all natural numbers in the interval $[\beta_{j+1}, \beta_j]$. Clearly $s \in A_j$ iff the access interval $\mathbf{A}_s$ is between the bookkeeping intervals $\mathbf{B}_{\beta_{j+1}}$ and $\mathbf{B}_{\beta_j}$. In the following lemma, as before, we denote the input pair arriving in interval $\mathbf{A}_s$ by $\langle a_s, v_s \rangle$

Lemma 6 *Assume that $a \in n$ so that at the end of $\mathbf{B}_i = \mathbf{B}_{\beta_0}$ there exists a pair $\langle a, x \rangle$ at node u for a suitably chosen x. Then there exists a set of integers $H_a \subseteq \bigcup_{j=0}^{r-1} A_j$, with the following properties:*

(22) $|H_a \cap A_j| \leq 1$ *for all* $j = 0, 1, ..., r-1$

(23) *for all* $h \in H$, *we have* $u \in \texttt{branch}(\chi(a_h))$

(24) *there exists an integer $h \in H_a$ so that $\chi(a_h) = \chi(a)$*

(25) *For all $j = 0, 1, ..., r-1$, $h \in H_a \cap A_j$, let $\sigma_h = \texttt{branch}(\chi(a_h)) \cap L_{\psi_j}$. Then, for all $h \in H_a$ with $\sigma_h \neq u$, there exists an integer $g \in H_a$ so that $\texttt{leaf}(\chi(a_g)) < \sigma_h < \sigma_g$*

Definition. An evaluation pair $\langle a, v\rangle$ will be called an a-type pair. The fact that at the end of time interval I_0 there exists a unique a-type pair for each $a \in \{0, 1, ..., n-1\}$, implies that at the end of each bookkeeping interval $\mathbf{B}_j$ there exists also a unique a-type pair, that we will call the a-type pair at the end of $\mathbf{B}_j$.

Proof of Lemma 6. We construct a set H_a with the required properties. For each $j = 0, 1, ..., r-1$, let ρ_j be the node where the a-type pair is at the end of interval $\mathbf{B}_{\beta_j}$. Recall that $\kappa(\beta_j) = \psi_j$. By the definition of $\mathcal{T}$ during a bookkeeping interval we have that $\rho_j \in \texttt{branch}(\chi(a))$ for all $j \in r$. We claim that either $\rho_j \in L_{\psi_j}$ or $\rho_j = \rho_{j+1}$. The reason for this is that according to Proposition 7 if there is an interval $\mathbf{B}_\zeta$ with $\beta_{j+1} < \zeta < \beta_j$, so that the a-type pair moved during one of these intervals then it ended in a level L_l with $l < \psi(j)$ and so in the interval $\mathbf{B}_{\beta_j}$ it moved to $L_{\kappa(\beta_j)} = L_{\psi_j}$. On the other hand assume that the a-type pair did not move in any of these intervals. At the end of interval $\mathbf{B}_{\beta_{j+1}}$ the a-type pair was in level $L_{\kappa(\beta_{j+1})} = L_{\psi_{j+1}}$ or below it. Since $\beta_{j+1} < \beta_j$, this implies that the a-type pair did not move in the interval $\mathbf{B}_{\beta_j}$ either. Therefore $\rho_j = \rho_{j+1}$.

Let J be the set of all $j = 0, 1, ..., r-1$ so that $\rho_j \neq \rho_{j+1}$. For each $j \in J$ let $s(j)$ be the smallest integer in the set A_j so that $a_{s(j)} = a$. $\rho_j \neq \rho_{j+1}$ implies that such an integer $s(j)$ exists.

Let $H_a = \{s(j) \mid j \in J\}$. Conditions (22) hold since for each A_j if $|H \cap A_j|$ is not empty then it contains only the smallest element of a set. Condition (23) is an immediate consequences of the fact that the a-type pair always moves along the branch $\texttt{branch}(\chi(a))$.

Condition (24). At the end of $\mathbf{B}_{\beta_r} = \mathbf{B}_{\alpha n 2^{-\mu}}$ the a type elements was in $L_{\kappa(\beta_j)} = L_d$, that is it was at a leaf. Assume that it moved from here in interval $A_{s(j)}$. Then $\chi(s(j)) = \chi(a)$.

Condition (25). Suppose that a $h \in H_a$ is given with $\sigma_h \neq u$, and $h = s(j) \in A_j \cap J$. By the definition of $s(j)$ we have $\rho_j \neq \rho_{j+1}$ and so $\rho_j \in L_{\kappa(\beta_j)} = L_{\psi_j}$ and clearly $\rho_j \in \texttt{branch}(a_{s(j)})$. This implies $\rho_j = \sigma_j$. Since $\sigma_h \neq u$ the a-type pair must be moved in an access interval from the node $\sigma_h = \rho_j$ therefore J' has at least one element j' with $j' > j$. Let j' be the smallest element of J with this property and let $g = s(j')$. Since the a-type pair is moved from $\rho_j = \sigma_h$ in interval $\mathbf{A}_g$ we have $\texttt{leaf}(\chi(a_g)) \leq \sigma_j < \sigma_g$. $\psi_l < d$ for $l = 0, 1, ..., l-1$ and $\sigma_j \in \bigcup_{l=0}^{r-1} L_l$ implies that $\sigma_j \notin L_d$, so we have $\texttt{leaf}(\chi(a_g)) < \sigma_j < \sigma_g$. *Q.E.D.*(Lemma 6)

Definition. 1. Assume that n, d, s are positive integers, $n = 2^d$, $\mathcal{L}$ is a set whose elements are levels of T_n, that is, $\mathcal{L} = L_{\varphi_0}, ..., L_{\varphi_{s-1}}$ where $\varphi_0 < ... < \varphi_{s-1}$. In this case we will say that $\mathcal{L}$ is a level-set with parameters $\varphi_0 < ... < \varphi_{s-1}$. In the following definitions we assume that such a level set $\mathcal{L}$ is fixed.

2. A set $K \subseteq T_n \times L_d$ is called an $\mathcal{L}$-tree, if the following conditions are satisfied:

(26) *for all $\langle \tau, a\rangle \in K$, we have $\tau >_{T_n}> a$*

(27) *there exists exactly one $\tau \in L_{\varphi_0}$ so that for a suitably chosen $a \in L_d$ we have $\langle \tau, a\rangle \in K$*

(28) *for all $\langle \tau, a\rangle \in K$, either $\tau \in L_{\varphi_0}$ or there exists a $\langle \sigma, b\rangle \in K$ with $b <_{T_n} \tau <_{T_n} \sigma$*

The unique element $\tau \in L_{\varphi_0}$ with the property in condition (27) will be denoted by 1_K

Lemma 7 *Assume that we choose λ in the definition of χ with $\lambda \geq 2$. Then for each fixed input sequence $\langle a_j, v_j\rangle$, $1 \leq j \leq n^{\log n}$, and for the randomization of χ, the following holds with a probability of at least $1 - n^{-3\log n}$. Suppose that there exist a $u \in T_n \backslash \mathbf{L}_\mu$ and an $i \in [1, n\log n]$ so that the number of evaluation pairs at the end of the bookkeeping interval $\mathbf{B}_i$ at node u is at least M_0. (We assume that, r, $\psi_0, ..., \psi_{r-1}$, $A_0, ..., A_{r-1}$ are defined from u and i as described before Proposition 6.) Suppose further that $\mathcal{L} = L_{\psi_0} \cup ... \cup L_{\psi_{r-1}}$ is the level set with parameters $\psi_0, ..., \psi_{r-1}$. Then*

(29) *there exists a set $Z \subseteq \bigcup_{j=0}^{r-1} A_j$ with $d^{-3}M_0 \leq |Z| \leq dM_0$ so that if for each $s \in A_j \cap Z$, $j \in r$, τ_s is the unique element of $\texttt{branch}(\chi(a_s)) \cap L_{\psi_j}$, and $b_s = \texttt{leaf}(\chi_a(s))$, then $K = \{\langle \tau_s, b_s\rangle \mid s \in Z\}$ is an $\mathcal{L}$-tree.*

Proof. Let W' be the set of all $a \in n$, so that there exists an a-type evaluation pair at the end of $\mathbf{B}_i$ at node u, and let W be a subset of W' with exactly M_0 elements. By the assumptions of the lemma such a set W exists. For each fixed $a \in W$ Let $H_a \subseteq \bigcup_{j=0}^{r-1} A_j$ be a set with the properties stated in Lemma 6. The existence of such a set H_a is guaranteed by that lemma. We define Z by $Z = \bigcup_{a\in W} H_a$. Clearly $|Z| \leq |W| \max_{a\in W} H_a$. By the assumptions of the present lemma we have $|W| = M_0$. $H_a \subseteq \bigcup_{j=0}^{r-1} A_j$ and condition (22) of Lemma 6 imply that $|H_a| \leq r \leq d$, therefore $|Z \leq d|M_0|$ as claimed..

To get a lower bound on $|Z|$ we use condition (24) of Lemma 6. The assumption $\lambda \geq 2$ together with Proposition 4 implies that with a probability of at least $1 - n^{-4\log n}$ we have that for each leaf b of T_n, the number of all $a \in n$ with $\texttt{leaf}(\chi(a)) = b$ is at most d_2. Lemma 6 implies that $\{\texttt{leaf}(\chi(a)) \mid a \in W \in Z\}$ has at least M_0^2/d^2 elements. By condition (24) of Lemma 6 each of these elements occur in a set $G_a = \{\texttt{leaf}(\chi(g)) \mid g \in H_a\}$ for some $a \in Z$. Since a single G_a has at most d elements we have $|Z| \geq d^{-3}M_0$.

Finally we have to show that K is an $\mathcal{L}$-tree. First we show that for each fixed $a \in W$, $K_a = \{\langle \tau_s, b_s\rangle \mid s \in Z\}$ is an $\mathcal{L}$-tree. By the definition of τ_s and b_s, both of them are on $\texttt{branch}(\chi(a_s))$, b_s is a leaf, and since $\psi_j < d$ for all $j = 0, 1, ..., r-1$, τ_s is not a leaf. Therefore $b_s < \tau_s$ and so K satisfies condition from the definition of a tree. Conditions (27), (28) of the definition is a consequence of condition (23) and (25) of Lemma 6.

K, the union of the $\mathcal{L}$-trees K_a, $a \in Z$ will be again a $\mathcal{L}$ tree since condition (27) of the definition of an $\mathcal{L}$-tree is satisfied in each of them with $\tau = u$. The other two conditions of the definition are trivially inherited for unions. *Q.E.D.*(Lemma 7)

Proof of Lemma 5. We will use the notation of Lemma 7. Assume that a $u \in T_n\mathbf{L}_\mu$ and an $i \in [1, n\log n]$ is fixed, and we randomize χ. We want to show that the probability of the event $B_{u,i,\gamma}$ is close to one, where $B_{u,i,\gamma} \equiv$ "the number of evaluation pairs at u at the end of the bookkeeping interval $\mathbf{B}_i$ will remain below d^γ". ($\gamma > 0$ is a constant that we will pick later.) Lemma 7 shows with $M_0 = d^\gamma$, that if $B_{u,i,\gamma}$ does not hold then there is a set $Z \subseteq A = \bigcup_{j=0}^{r-1} A_j$ so that $|Z|$ is around $M_0 = d^\gamma$, and $K = \{\langle \tau_s, b_s\rangle \mid s \in Z\}$ is a K-tree. We will prove the statement of Lemma 5 by showing that for the randomization of χ the probability that A has such a subset Z is negligible.

We reformulate this statement. For each $s \in A = \bigcup_{i=0}^{r-1}$ let ξ_s be a random variable defined in the following way. Suppose that u, i are fixed. They uniquely determine r, $A_0, ..., A_{r-1}$, $\psi_0, ..., \psi_r - 1$, $\mathcal{L} = \bigcup_{j=0}^{r-1} L_{\psi_j}$. We randomize now ψ with $\lambda > \gamma + 1$, $\lambda \geq 2$. The value of ξ_s is the pair $\langle \tau_s, b_s \rangle$ where, if $s \in A_j$, τ_s is the unique element of $\texttt{branch}(\chi(a_s)) \cap L_{\psi_j}$, and $b_s = \texttt{leaf}(\chi_a(s))$. The Lemma will 8 below will show that the probability that the values of such random variables form an $\mathcal{L}$-tree of the required size is very low.

The following Lemma 8 is completely self-contained, that is, we do not assume that $r, \psi_0, ..., \psi_{r-1}$, $A_0, ..., A_{r-1}$, ξ_x are defined as in the previous lemmas or in the explanation above, they are arbitrary integers sets and random variables with the properties only stated in the lemma. We will apply the lemma however in a way when these elements will play the roles indicated previously. This more general formulation of the lemma, perhaps shows, what is the combinatorial problem that we have to solve for the proof of Lemma 5.

Lemma 8 *Assume that d, n are positive integers with $n = 2^d$, $0 < \alpha < 1$, k is a positive integer, and $\mathcal{L} = \{L_{\psi_0}^{(T_n)}, ..., L_{\psi_{r-1}}^{(T_n)}\}$ is a level-set of the tree T_n with parameters $\psi_0 < ... < \psi_{r-1} < d$. Suppose further that an $u \in L_{\psi_0}^{(T_n)}$ is fixed and $A_0, ..., A_{r-1}$ are pairwise disjoint sets and for each $a \in A = \bigcup_{i=0}^{r-1} A_i$, $\xi_a$is a random variable with the following properties:*

(30) $|A_i| \leq \alpha |L_{\psi_i}| = \alpha 2^{\psi_i}$ *for all* $i = 0, 1, ..., r-1$

(31) *For each $i = 0, 1, ..., r-1$ and $a \in A_i$, the random variable ξ_a takes its values on $L_{\psi_i}^{(T_n)} \times L_d$ with uniform distribution on the set of all pairs $\langle \tau, b \rangle \in L_{\psi_i} \times L_d$ with the property $\tau \geq_{T_n} b$.*

(32) *Suppose $Y \subseteq A$ and $|Y| \leq k$. Then the random variables in ξ_a, $a \in Y$ are mutually independent.*

Let X be the set of all random variables ξ_a, $a \in A$. We randomize all of the random variables in X. Then the probability of the following event is at most $(\alpha d^2 k)^k$:

(33) *There exists a $Z \subseteq A$ so that $|Z| = k$, and if K is the set of values of the random variables ξ_a, $a \in Z$ then (a) K is an $\mathcal{L}$-tree, and (b) for each $\langle \tau, b \rangle \in K$ we have $\tau \leq_{T_n} u$.*

We will need the following definitions in the proof of the lemma.

Definition. Suppose that K is an $\mathcal{L}$-tree. We define a directed graph on K. An edge points from $\langle \tau_1, b_1 \rangle$ to $\langle \tau_2, b_2 \rangle$ iff $b_2 < \tau_1 < \tau_2$. We will denote this directed graph by $\texttt{graph}(K)$.

Remark. The directed graph $\texttt{graph}(K)$ does not contain circles, since along a path $\langle \tau_i, b_i \rangle$, $i = 1, 2, ...$, the elements τ_i form a strictly increasing sequence in the ordering $\leq_{T_n}$. The definition of an $\mathcal{L}$-tree also implies that if there is no outgoing edge from an element $\langle \tau, b \rangle \in K$, then $\tau \in L_{\psi_0}$. Therefore condition (28) implies that any directed path in $\texttt{graph}(K)$ can be extended into a path which ends in an element $\langle \tau, b \rangle$ with $\tau \in L_{\psi_0}$.

Definition. Assume that K is an $\mathcal{L}$-tree and F is a function defined on the totally ordered set Γ with $\texttt{range}(F) = K$. The ordering on Γ will be denoted by $\leq_\Gamma$. $\texttt{rank}_\Gamma(x)$ will denote the integer $|\{y \in \Gamma \mid y <_\Gamma x\}|$. We define a function φ on the set $|\Gamma| = \{0, 1, ..., |\Gamma| - 1\}$ with values in $|\Gamma| \times r$. We define first two functions $v(i)$ and $s(i)$ on $|\Gamma|$, and then we define φ by $\varphi(i) = \langle v(i), s(i) \rangle$ for all $i \in |\Gamma|$. Suppose that $i \in |\Gamma|$ and $\gamma \in \Gamma$, $\mathtt{rank}_\Gamma(\gamma) = i$. Assume that $F(\gamma) = \langle \tau, b \rangle$. $s(i)$ is the unique integer with $\tau \in L_{\psi_{s(i)}}$. If there is no outgoing edge in $\mathtt{graph}(K)$ from $F(\gamma)$ then $v(i) = 0$. Otherwise let δ be the smallest element of Γ with respect to $\leq_\Gamma$, with the property that $b' < \tau < \sigma$, where $F(\delta) = \langle \sigma, b' \rangle$. According to condition (28) of the definition of an $\mathcal{L}$-tree such a δ always exists. We define $v(i)$ by $v(i) = \mathtt{rank}_\Gamma(\delta)$. The function φ defined above will be called the type of the function F. A type defined for functions F with $\mathtt{domain}(F) = k$, will be called a k-type

Remark. The type of F, if F is a function defined on Γ with $\mathtt{range}(F) = K$, is a map from $|\Gamma|$ into $r \times |\Gamma|$. Therefore the number of possible types of functions F defined on Γ, with values in K is at most $(r|\Gamma|)^{|\Gamma|}$. Therefore the number of k-types is at most $(rk)^k$

Definition. 1. $\mathtt{func}(X, Y)$ will denote the set of all functions f with $\mathtt{domain}(f) = X$ and $\mathtt{range}(f) \subseteq Y$.

2. Suppose that f is a function, $a \notin \mathtt{domain}(f)$ then $f\wr_{a,b}$ will denote the unique extension of f to the set $\mathtt{domain}(f) \cup \{a\}$ defined by $f(a) = b$.

Definition. Assume that $\langle u_0, v_0 \rangle, ..., \langle u_{s-1}, v_{s-1} \rangle$ is a sequence of pairs. We say that the sequence is one-to-one if (a) the elements $u_0, ..., u_{s-1}$ are distinct and (b) the elements $v_0, ..., v_{s-1}$ are also distinct. (It is possible that $u_j = v_j$ for some $i, j \in s$.)

Lemma 9 *Suppose that $\beta > 0$ is a real, and the following conditions are satisfied:*

(34) *k is a positive integer, J, H, W are finite sets, and $|J| = k$*

(35) *R is a symmetric k-ary relation on the set $J \times H$ so that $R(\langle \iota_0, h_0 \rangle, ..., \langle \iota_{k-1}, h_{k-1} \rangle)$ implies that $\langle \iota_0, h_0 \rangle, ..., \langle \iota_{k-1}, h_{k-1} \rangle$ is a one-to-one sequence.*

(36) *for each $w \in W$, η_w is a random variable, which takes its values on the set H, and for any subset Y of W with $|Y| \leq k$, the random variables η_w are independent.*

(37) *Assume that for each $j \in [0, k-1)$, and for each one-to-one sequence $\langle \iota_0, h_0 \rangle, ..., \langle \iota_{j-1}, h_{j-1} \rangle$ there exists an $\iota_j \in J \backslash \{\iota_0, ..., \iota_{j-1}\}$, so that we have $\sum_{w_j \in W} q_{w_j, \iota_j} < \beta$, where q_{w_j, ι_j} is the probability of the following event Q_{w_j, ι_j} with respect to the randomization of η_{w_j}:*

There exist $h_{j+1}, ..., h_{k-1} \in H$ and $\iota_j, \iota_{j+1}, ..., \iota_{k-1} \in J$ so that

$$R(\langle \iota_0, h_0 \rangle, ..., \langle \iota_{j-1}, h_{j-1} \rangle, \langle \iota_j, \eta_{w_j} \rangle, \langle \iota_{j+1}, h_{j+1} \rangle, ..., \langle \iota_{k-1}, h_{k-1} \rangle)$$

Then the probability of the following event is at most β^k. There exists a sequence $\langle i_0, v_0 \rangle, ..., \langle i_{k-1}, v_{k-1} \rangle$ so that $i_0, ..., i_{k-1}$ is a permutation of the elements J, $v_0, ..., v_{k-1} \in W$, and

$$R(\langle i_0, \eta_{v_0} \rangle, ..., \langle i_{k-1}, \eta_{v_{k-1}} \rangle)$$

Proof. We prove the lemma by induction on k. For $k=1$ the assumptions of the lemma imply that $\sum_{w\in W}\texttt{prob}(R(\langle 0,\eta_w\rangle))<\beta$. Clearly the sum in this expression is also an upper bound on $\texttt{prob}(D_R)=\texttt{prob}(\exists w\in w, R(\langle 0,\eta_w\rangle))$ which implies our statement.

Assume now that the lemma holds for $k-1$ and we prove it for k. According to the assumptions of the lemma with $j=1$ there exists a $\iota_0\in J$ so that $\sum_{w\in W} q_{w,\iota_0}<\beta$. Let B be the event in the conclusion of the lemma, and for each $w\in W$ let $B_{\iota_0,w}$ be the event where in the definition of B we include the additional requirement for each $s=0,...,k-1$, $i_s=\iota_0$ implies $w_s=w$.

In the definition of B $i_0,...,i_{k-1}$ is a permutation of J, therefore there is always an $s\in k$ with $i_s=\iota_0$, and in this case B implies B_{ι_0,w_s}. Hence we have $\texttt{prob}(B)\le\sum_{w\in W}\texttt{prob}(B_{\iota_0,w})$. We randomize η_w. With a probability of $1-q_{w,\iota_0}$ we get a value η_w which guarantees that $B_{\iota,w}$ does not hold. We use Bayes theorem for the randomization of η_w we get $\texttt{prob}(B_{\iota_0,w}=\sum_{h\in H_w}\texttt{prob}(\eta_w=h)\texttt{prob}(B_{\iota_0,w}|\eta_w=h)$, where H_w is the set of all $h\in H$ with $\texttt{prob}(\eta_w=h)\neq 0$. Let H'_w be the set of all $h\in H_w$ with $\texttt{prob}(B_{\iota_0,w}|\eta_w=h)>0$. According to our assumptions $\sum_{w\in W}\texttt{prob}(\eta_w\in H'_w)<\beta$.

Therefore it is sufficient to show that for all $h\in H'_w$ we have $\texttt{prob}(B_{\iota_0,w}|\eta_w=h)\le\beta^{k-1}$. Indeed, this would imply $\texttt{prob}(B)\le\sum\texttt{prob}(B_{\iota_0,w})\le\sum_{w\in W}\sum_{h\in H'_w}\texttt{prob}(\eta_w=h)\texttt{prob}(B_{\iota_0,w}|\eta_w=h)\le\sum_{w\in W}\sum_{h\in H'_w}\texttt{prob}(\eta_w=h)\beta^{k-1}\le\beta^{k-1}\sum_{w\in W}\texttt{prob}(\eta_w\in H'_w)\le\beta^k$.

We show now that for each fixed $\tilde{w}\in W$, $h\in H'_{\tilde{w}}$ we have $\texttt{prob}(B_{\iota_0,\tilde{w}}|\eta_{\tilde{w}}=h)\le\beta^{k-1}$. We apply the inductive assumption with $\bar{k}=k-1$, $\bar{J}=J\backslash\{\iota_0\}$. The sets W, H remain unchanged, and the $k-1$ ary relation $\bar{R}$ will hold on the sequence $\langle i_0,g_0\rangle,...,\langle i_{k-2},g_{k-2}\rangle$ iff the k-ary relation R holds on the sequence $\langle\iota_0,h\rangle,\langle i_0,g_0\rangle,...,\langle i_{k-2},g_{k-2}\rangle$. For each $w\in W$ will $\bar{\eta}_w$ will be a random variable whose distribution is the conditional distribution of η_w with the condition $\eta_{\tilde{w}}=h$.

We have to show that the conditions (34), (35), (36), (37) hold for these values as well. The first two conditions are immediate consequences of the definitions. For the proof of condition (36) let $w_0,...,w_{k-2}\in H$. Condition (36) for the original k-dimensional case implies that $\eta_{w_0},...,\eta_{w_{k-2}},\eta_k$ are independent. Therefore if we consider the first $k-1$ of these random variables conditioned with an event about the last one, then they remain independent.

Finally we prove that condition (37) is satisfied. The statement for the relation $\bar{R}$ is simply a special case for the corresponding statement about the relation R since the independence of $\eta_{\tilde{w}}$ and η_{w_j} guaranteed by (36) for the k-dimensional statement implies that the distributions of $\bar{\eta}_{w_j}$ and η_{w_j} are the same. *Q.E.D.*(Lemma 9)

Proof of Lemma 8. We fix a total ordering $\le_A$ of the set A. Assume that the random variables ξ_a has been randomized and condition (33) is satisfied. Let F be the function which assigns the value of ξ_a to the integer $\texttt{rank}_Z(a)\in k$. F is a function whose range is an $\mathcal{L}$-tree. Let φ be the type of F under the ordering of Z induced by $\le_A$. The element of a of Z with $\texttt{rank}(a)=i$ will be denoted by z_i.

For each k-type φ_0 we estimate the probability p_{φ_0} that condition (33) holds with $\varphi=\varphi_0$. Since the number of k-types is at most $(rk)^k$ this will give an estimate on the probability that condition (33) is satisfied (without any restrictions).

Assume now that a k-type φ_0 is fixed. We apply Lemma 9, with $\beta=\frac{1}{d}k$, $J=\{0,1,...,k-1\}$, $H=(\bigcup_{i\in r}L_{\psi_i})\times L_d$, $W=A$. The symmetric relation R is defined as follows: assume that

$\iota_i \in k$, $h_i \in H$ for $i \in k$. $R(\langle \iota_0, h_0\rangle, ..., \langle \iota_{k-1}, h_{k-1}\rangle)$ holds iff $\iota_0, ..., \iota_{k-1}$ is a permutation of the elements of k and if the map F defined by $F(\iota_i) = h_i$ for all $i \in k$ has type φ_0, where we consider the natural ordering on k.

Finally we will have $\eta_a = \xi_a$ for all $a \in A$.

We have to show that conditions (34), (35), (36), and (37) are satisfied by these choices of the parameters. The first three of these conditions are immediate consequences of the definitions.

Condition (37). Assume that $\langle \iota_0, h_0\rangle, ..., \langle \iota_{j-1}, h_{j-1}\rangle$ is a one-to-one sequence so that $\iota_0, ..., \iota_{j-1} \in J = k$, and $h_0, ..., h_{j-1} \in H = (\bigcup_{i \in r} L_{\psi_i}) \times L_d$. Now we have to select a ι_j from k with the properties described in the condition (37). ι_j must be in the set $k \backslash \{\iota_0, ..., \iota_{j-1}\}$. For each element $\iota \in S$ we consider the pair $\varphi_0(\iota) = \langle v(\iota), s(\iota)\rangle$. ι_j will be an arbitrary element of S so that $s(\iota_j)$ is maximal, that is, for all $\iota \in S$, $s(\iota_{j_0}) \geq s(\iota)$.

We have to prove now the inequality $\sum_{w_j \in A} q_{w_j, \iota_j} < \alpha$. Since $s(\iota_j)$ is maximal, the definitions of an $\mathcal{L}$-tree and a k-type imply that either (a) $s(\iota_j) = \psi_0$, or (b) there exists an $x \in j$ with $v(\iota_j) = \iota_x$.

Consider first the case when condition (b) holds. Assume that w_j is fixed, if the conclusion for (37) holds then ι_i, h_i can be defined for $i = j, ..., k-1$ so that the map $F(\iota_i) = h_i$, $i \in k$ has type φ_0 and $h_j = \xi_{w_j}$. This would imply that if $h_x = \langle \sigma, b\rangle$ and $\xi_{w_j} = h_j = \langle \tau, b'\rangle$, then $\sigma > \tau \geq b$. Let y be the unique element of T_n so that $\sigma > y > b$ and $y \in L_{s(\iota_j)}$. Clearly $y = \tau$.

Therefore the random variable ξ_{w_j} must take a value $\langle y, b\rangle$ otherwise the conclusion of (37) cannot hold. This is a trivial consequence of condition (a) as well with $y = u = 1_K$. Therefore from now on we can consider cases (a) and (b) together.

This happens with nonzero probability only of $w_j \in L_m$, where $m = s(\iota_j)$, and in this case according to condition (30) of Lemma 8 $w_j \in A_m$ the probability of $\xi_{w_j} = \langle y, b'\rangle$ for a suitably chosen b' is $|L_{\psi_m}|^{-1}$. On the other hand according to condition (30) of Lemma 8 we have $|A_i| \leq \alpha |L_m|$. Therefore $\sum_{w_j \in A} q_{w_j, \iota_j} = \sum_{w_j \in A_m} q_{w_j, \iota_j} \leq \alpha |L_m| |L_m|^{-1} \leq \alpha$ which completes the proof of condition (37)

We have shown that Lemma 9 is applicable with the described values of the parameters. The conclusion of Lemma9 in the present case is the following.

With a probability of at least $1 - \alpha^k$ for the randomization of all of the random variables ξ_a, $a \in A$, it is not possible to select $v_0, ..., v_{k-1} \in A$ so that the type of the function F, defined by $F(i) = \xi_{v_i}$, is φ_0. This is true for each k-type φ_0. The number of k-types is at most $(kd)^k(d)^k$. Therefore the probability that there exists a set $Z \subseteq W$ is at most $\alpha^k(d^2k)^k = (\alpha d^2 k)^k$. *Q.E.D.*(Lemma 8)

Proof of Lemma 5 continued. Now we can apply Lemma 8 for the integers $r, \psi_0, ..., \psi_{r-1}$, and the sets $A_0, ..., A_{r-1}$, that were used in Lemma 7 and defined before the statement of Proposition 6. We use the random variables ξ_s, $s \in \bigcup_{j=0}^{r-1} A_j$ as they were defined before the statement of Lemma 8. Let k be an arbitrary but fixed integer in the interval $[d^{\gamma-3}, d^{\gamma+1}]$, and let $\alpha = 2^{-\mu} \leq d^{-c_1}$. Then the assumption of Lemma 8 are satisfied, and so Lemma 8 implies that the probability p for the randomization of χ that condition (29) of Lemma 7 holds with $k = M_0$ is at most $(\alpha d^2 k)^k \leq (d^{-c_1} d^2 k)^k$. If we pick c_1 with $c_1 > \gamma + 4$ then we get $p \leq d^{-k} \leq d^{-d^{\gamma-3}}$. Therefore it $\gamma \geq 5$ and $c_1 \geq \gamma + 4$ and $\lambda \geq \gamma + 3$ then we have $p \leq n^{-\log n \log\log n}$. Therefore for $M_0 = \lceil d^\gamma \rceil$, and for all fixed choices of i, u with $i \in \{1, n^{\log n}\}$, $u \in T_n$ using Lemma 7 we get that the probability that there are at least M_0 evaluation pairs at u at the end of interval $\mathbf{B}_i$ is at most $n^{-3\log n} + n^{-\log n \log\log n}$. Since the number of choices for u and i is at most $n^{1+\log n}$

this completes the proof of the lemma. *Q.E.D.*(Lemma 5)

4 The algorithm $\mathcal{A}$

We want to define the algorithm $\mathcal{A}$ in a way that its input/output behavior is identical to the algorithm $\mathcal{T}$ and at the same time it is $\tilde{\gamma}$-oblivious. We will define $\mathcal{A}$ by translating $\mathcal{T}$ into an algorithm which works on $\mathcal{M}_q$.

Motivation. As we explained in the sketch of the proof, the algorithm $\mathcal{A}$ will search (the same way as algorithm $\mathcal{Y}$) for a pair of the form $\langle a_i, v\rangle$ in the interval $\mathbf{A}_i$. Suppose that a $k = 1, ..., d$ is fixed, and τ is the unique element of $L_k \cap \mathtt{branch}(\chi(a_i))$. In the subinterval $\mathbf{A}_{i,k}$ the following happens.

Case I. The coupon $\mathbf{c}(\tau)$ is *not* in the tray at the beginning of $\mathbf{A}_{i,k}$. In this case $\mathcal{A}$ searches a random bucket $\mathbf{b}$, with the only purpose of deceiving the adversary. The bucket $\mathbf{b}$ will be selected by the random variable $\bar{\zeta}_{i,k}$, to be defined below.

Case II. The coupon $\mathbf{c}(\tau)$ is in the tray at the beginning of $\mathbf{A}_{i,k}$. Then $\mathcal{A}$ searches for $\langle a_i, v\rangle$ in the bucket $\mathbf{p}(\tau)$. The bucket $\mathbf{p}(\tau)$ was selected at random in an earlier time interval $\mathbf{B}_j$. In the following definition we define a random variable $\zeta_{i,\sigma}$ that determines the results of this selection (with a different value for the parameter i).

Summarizing the two cases from the point of view of the adversary: the algorithm $\mathcal{A}$ searches a bucket. This bucket will be determined by the random variable $\eta_{i,k,\mathcal{I}}$, where $\mathcal{I}$ is the input sequence, where $\eta_{i,k,\mathcal{I}}$ is defined from the values of the random variables $\zeta_{i,k}$ and $\bar{\zeta}_{i,k}$. $\mathcal{A}$ will be defined in a way that the only thing that the adversary will learn from the visible history (in addition to the conceded history) is the value sequence of the random variable $\eta_{i,k,\mathcal{I}}$. Therefore our goal will be to show that the random variables $\eta_{i,k,\mathcal{I}}$ are mutually independent, their distributions are predetermined (independently of the input), and if the values of the random variables $\eta_{i,k,\mathcal{I}}$ are fixed, then the visible history is uniquely determined. In other words with a fixed sequence $\eta_{i,k,\mathcal{I}}$, $\mathcal{A}$ is oblivious in a deterministic sense.

Definition. In the following definitions we assume that a constant $c_1 > 0$ is fixed for the defintion of $\mathcal{T}^{(\chi,c_1)}$.

1. We define $\kappa(0)$ by $\kappa(0) = d$.

2. Assume that m is a positive integer. For all $i = 0, 1, ..., m$, $\sigma \in \mathbf{L}_{\kappa(i)}$, we define a random variable $\zeta_{i,\sigma}$. Let j be the unique integer with $\sigma \in L_j$. By definition, $\zeta_{i,\sigma}$ is a random variable with uniform distribution in $\{0, 1, ..., 2^j - 1\}$. We also define a random variable $\bar{\zeta}_{i,k}$ for all $i = 0, 1, ..., m$ and $k = 0, 1, ..., d$. $\bar{\zeta}_{i,k}$ is a random variable with uniform distribution in $\{0, 1, ..., 2^k - 1\}$. We assume that all of the random variables $\zeta_{i,\sigma}$, $\bar{\zeta}_{i,k}$, and the random variable χ, used for the selection of the algorithm $\mathcal{T}^{(\chi,c_1)}$, together are mutually independent.

3. Assume that $\mathcal{I} = \langle\langle a_i, v_i, \Psi_i\rangle \mid i = 1, ..., m\rangle$ is an input sequence for $\mathcal{T}$, that is, $a_i \in \{0, 1, ..., n-1\}$, $v_i \in [0, \wp]$, $\Psi_i \in \{0, 1\}$ for $i = 1, ..., m$. We randomize all of the random variables $\zeta_{i,\sigma}, \bar{\zeta}_{i,k}$ and χ, for $i = 0, 1, ..., m$, $\sigma \in \mathbf{L}_{\kappa(i)}$, and $k = 1, ..., d$. Depending on the input $\mathcal{I}$ and the results of these randomizations we define the integers $\eta_{i,k,\mathcal{I}}$, $i = 1, ..., m$, $k = 1, ..., d$ in the following way. Suppose that a pair i, k is given with $i \in \{1, ..., m\}$, $k \in \{0, 1, ..., d\}$. To determine the value of $\eta_{i,k,\mathcal{I}}$ we consider two separate cases.

Assume the algorithm $\mathcal{T}^{(\chi,c_1)}$ is executed with the function χ provided by the randomization, and with the input $\mathcal{I}$. The unique element of the set $L_k \cap \texttt{branch}(\chi(a_i))$ will be denoted by $\tau(i,k)$.

Case I. At the beginning of interval $\mathbf{A}_{i,k}$ the coupon $\mathbf{c}(\tau(i,k))$ is not in the tray. In this case the value of $\eta_{i,k,\mathcal{I}}$ is $\bar{\zeta}_{i,k}$.

Case II. At the beginning of the interval $\mathbf{A}_{i,k}$, the coupon $\mathbf{c}(\tau(i,k))$ is in the tray. In this case let $j(i,k)$ be the largest nonnegative integer so that $j(i,k) < i$ and $\kappa(j(i,k)) \geq k$. (Since $\kappa(0) = d$ such an integer always exists.) The value of $\eta_{i,k,\mathcal{I}}$, by definition, is $\zeta_{j(i,k),\tau(i,k)}$.

Lemma 10 *Assume that $c_1 > 0$, $\mathcal{I} = \langle\langle a_i, v_i, \Psi_i\rangle \mid i = 1,...,m\rangle$ is an input sequence for $\mathcal{T}$, we randomize all of the random variables $\zeta_{i,j}, \bar{\zeta}_{i,k}$ and χ, for $i = 0,1,...,m$, $j = 1,...,\kappa(i)$, and $k = 1,...,d$. We consider $\eta_{i,k,\mathcal{I}}$, $i = 1,...,m$, $k = 1,...,d$ with respect to this randomization as random variables. Then the random variables $\eta_{i,k,\mathcal{I}}$, $i = 1,...,m$, $k = 1,...,d$, and χ are mutually independent, and $\eta_{i,k,\mathcal{I}}$ has uniform distribution on $\{0,1,...,2^k - 1\}$.*

Proof. First we randomize χ and assume that we get the function χ_0. It is sufficient to show that for each possible choice of the function χ_0, and the input $\mathcal{I}$, the conditional distribution of the random variables $\eta_{i,k}$ $i = 1,2,...,m$, $k = 1,...,d$ with the condition $\chi = \chi_0$ are mutually independent and the conditional distribution $\eta_{i,k}$ is uniform on $\{0,1,...,2^k - 1\}$.

First we note that the values of $\mathcal{I}$ and χ uniquely determine whether in the definition of $\eta_{i,k}$, we have Case I or Case II, and if Case II must be considered what is the value of $j(i,k)$. This is true because these questions are determined by the history of $\mathcal{T}$ and the values of χ and $\mathcal{I}$ (together with c_1 which has been fixed) uniquely determine the history of $\mathcal{T}$.

For each fixed $k = 1,...,d$ let H_k be the set of all $i = 1,....,m$ so that in the definition of $\eta_{i,k}$, Case II holds. We claim that for $i, i' \in H_k$, if $i \neq i'$ then $\langle j(i,k), \tau(i,k)\rangle \neq \langle j(i',k), \tau(i',k)\rangle$. If $\tau(i,k) \neq \tau(i',k)$ then this trivially holds. Assume that $\tau(i,k) \neq \tau(i',k) = \tau$ and e.g., $i < i'$. According to the definition of Case II, at the beginning of interval $\mathbf{A}_{i,k}$, the coupon $\tau = \mathbf{c}(\tau(i,k))$ is in the tray. Then, as described in step (12) in the definition of the action of $\mathcal{T}$ in the interval $\mathbf{A}_{i,k}$, the coupon $\mathbf{c}(\tau)$ is destroyed. Since in the interval $\mathbf{A}_{i'}$ the coupon $\mathbf{c}(\tau)$ is active, it must have been reactivated in an epoch $\mathbf{B}_r$ with $i \leq r < i'$. Since $\tau \in L_k$ and τ is reactivated in $\mathbf{B}_r$, by the defintion of $\mathcal{T}$ in $\mathbf{B}_r$, we have that $\tau \in L_k \subseteq \mathbf{L}(\kappa(r))$ and so $k \leq \kappa(r)$. This, together with $j(i,k) < i \leq r < i'$ and the definition of $j(i',k)$, implies that $j(i',k) \geq i$ and so $j(i,k) \neq j(i',k)$ as claimed.

We have proved that for a fixed $k = 1,...,d$, all of the random variables $\zeta_{j(i,k),\tau(i,k)}$, $i \in H_k$ are different. Of course if $k \neq k'$ then $\tau(i,k) \in L_k$, $\tau(i,k') \in L_{k'}$ implies that $\tau(i,k)$ and $\tau(i,k')$ are different. Consequently all of the random variables $\zeta_{j(i,k),\tau(i,k)}$ that are used in the definition of the random variables $\eta_{i,k}$ through Case II are different.

Therefore the first statement of the lemma can be formulated in the following way. Suppose that depending on $\mathcal{I}$ and χ_0 we take a subset Z of the set of random variables $\zeta_{i,\sigma}$, $\bar{\zeta}_{i,k}$, $i = 0,1,...,m$, $\sigma \in \mathbf{L}_{\kappa(i)}$, $k = 1,...,d$. Then the conditional random variables in Z with the condition $\chi = \chi_0$ are mutually independent. This is true since the mutual independence of χ and all of the other random variables imply that all of the conditional random variables are independent and a subset of mutually independent random variables is also mutually independent. The second statement that $\eta_{i,k}$ has uniform distribution on 2^k follows from the fact that both $\zeta_{i,\sigma}$,

$\sigma \in L_k$ and $\bar{\zeta}_{i,k}$ have uniform distribution on 2^k, and they are independent of χ. Therefore their distribution remain the same with the condition $\chi = \chi_0$. *Q.E.D.*Lemma 10

We will define $\mathcal{A}$ in a way that $\mathcal{A}$ randomizes χ and $\zeta_{i,\sigma}$, $\bar{\zeta}_{i,k}$, $i = 0, 1, ..., m$, $\sigma \in \mathbf{L}_{\kappa(i)}$, $k = 1, ..., d$. The values of these random variables uniquely determine the values of the random variables $\eta_{i,k}$, $i = 1, ..., m$. We will define $\mathcal{A}$ so that if the values of $\eta_{i,k}$, $i = 1, ..., m$ are fixed, then $\mathcal{A}$ is $\tilde{\gamma}$-oblivious in a deterministic sense, that is, the access pattern of $\mathcal{A}$ (outside the first $\tilde{\gamma}$ memory cells) depend only on the values of $\eta_{i,k}$, $i = 1, ..., m$. As Lemma 10 states the random variables $\eta_{i,k}$ are mutually independent, and their distributions are determined by the value of k. This clearly implies the $\tilde{\gamma}$-obliviousness of $\mathcal{A}$.

Motivation. The next definitions are motivated by the fact, that if a deterministic algorithm uses only $(\log n)^c$ memory cells whose set M is fixed in advance, then it is easy to simulate it, so that the time is increasing only by about a factor of $(\log n)^c$. (This is used in [6], [8], [9], [7] repeatedly.) Indeed, every time when we have to access a memory cell, we access all of them but perform the required read or write operation only at the single place where it is needed. Based on this observation, we define the algorithm $\mathcal{A}$ in two steps. First we define an algorithm $\mathcal{A}'$ whose time can be partitioned into intervals J_i, so that in each interval J_i the algorithm accesses only memory cells form a set M_i, where $|M_i| \leq (\log n)^c$, and the sequence M_i is uniquely determined by the values of the random variables $\eta_{i,k,\mathcal{I}}$. In the next step, in each interval J_i we replace $\mathcal{A}'$ with an oblivious algorithm as indicated above. *End of motivation.*

First we define another algorithm $\mathcal{A}'$ and will get $\mathcal{A}$ by modifying $\mathcal{A}'$. For the definition of $\mathcal{A}'$ we cut total the time, that is, the interval $I_0 \cup \bigcup_{i=1}^{m} \mathbf{A}_i \cup \mathbf{B}_i$, into subintervals $J_0, ..., J_{m'}$, each of length at most d^{α_1}, where $\alpha_1 > 0$ is a constant. We define this division in a way that it is uniquely determined by n, c_1, and m. To each time interval J_i we assign a set of memory cells M_i with the following properties:

(a) $|M_i| \leq d^{\alpha_1}$,

(b) M_i is uniquely determined by n, c_1, and the values of the random variables $\eta_{i,k,\mathcal{I}}$, $i = 1, ..., m$, $k = 1, ..., d$,

(c) the list of the addresses of the memory cells in M_i can be computed in time d^{α}, using only the first $\tilde{\gamma}$ memory cells and their contents at the beginning of time interval J_i.

We will define $\mathcal{A}'$ so that in the time interval J_i, $i = 1, ..., m'$ it will access memory cells only from M_i, apart from the first $\tilde{\gamma}$ memory cells. (It is also important that $\mathcal{A}'$ will be implemented by a program of constant size running on $\mathcal{M}_q$.)

Suppose that we have defined $\mathcal{A}'$ so that it has the same input/output behavior as $\mathcal{T}$ and satisfies the conditions described above. We get $\mathcal{A}$ from $\mathcal{A}'$ in the following way. In each time interval J_i we replace $\mathcal{A}'$ with an algorithm whose memory access pattern is uniquely determined by M_j. This can be done by simply replacing each access by $\mathcal{A}$ for a memory cell in M_i by M_i accesses of $\mathcal{A}$ to each of the memory cells in M_i. Since the list of memory cells can be computed in time d^{α} using only the first $\tilde{\gamma}$ cells, we get that $\mathcal{A}$ can perform this action altogether in time $d^{4\alpha}$. Such an algorithm α will be oblivious since its access pattern does not reveal anything else to the adversary then the sets M_i or equivalently the values of the random variables $\eta_{i,k}$.

The definition of the algorithm $\mathcal{A}'$. Each interval $\mathbf{A}_{i,k}$ will be also an interval J_j. We will describe later the divisions of the intervals $\mathbf{B}_i$ and I_0 into subintervals J_j.

Assume that $\gamma > 0$ and $\lambda > 0$ are the constants whose existences are guaranteed in Lemma 5. χ will be d^{λ}-wise independent. Assume that such a χ has been selected (the selection will be done by $\mathcal{A}'$ in interval I_0 that we will describe later).

The algorithm $\mathcal{A}'$ several times will choose a new value for $\mathbf{p}(\tau)$ for all $\tau \in T_n$ with $\mathbf{p}(\tau) \in \mathcal{B}_k$, if $\tau \in L_k$. If $\mathbf{p}(\tau) = \mathbf{b}_{k,j}$ then we will say that $\langle k, j \rangle$ is the (current) address of the bucket, and we will write $\bar{\mathbf{p}}(\tau) = \langle k, j \rangle$.

The randomization of $\zeta_{i,\sigma}$ for $i = 0, 1, ..., m$, and $\sigma \in L_{\kappa,i}$ will be done by $\mathcal{A}'$ in interval $\mathbf{B}_i$. After this randomization the new value of $\mathbf{p}(\sigma)$ will be the bucket $\mathbf{b}_{k,\zeta_{i,\sigma}}$ and for $\bar{\mathbf{p}}(\sigma)$ the pair $\langle k, \zeta_{i,\sigma} \rangle$. This definition is also valid for $i = 0$, that is for $\mathbf{B}_0 = I_0$ when the initial value of $\mathbf{p}(\sigma)$ is set. This way we defined the value of $\mathbf{p}(\sigma)$, $\bar{\mathbf{p}}(\sigma)$ while the algorithm is working. (This was just an abstract definition in itself it does not guarantee that any of these values are easily available for $\mathcal{A}'$.)

We reserve a block of poly$(\log n)$ consecutive cells on $\mathcal{M}_q$ where the information corresponding the content of the tray of $\mathcal{T}$ will be kept. We denote this block of cells by `tray`. `tray` is disjoint from all of the buckets and the set of first $\tilde{\gamma}$ memory cells.

Definition. 1. We assume that the elements of the tree T_n are encoded by elements of the set $\{0, 1, ..., 2n-2\}$, in some efficient way, in the sense that the relations on the tree can be efficiently computed using the encoding. The integer corresponding to the node τ will be denoted by $\bar{\tau}$.

2. Let $x \rightarrow \mathbf{b}[x]$ be a one-to-one map of the set $0, 1, ..., 2n-2$ into the set of all buckets with the property that if $\tau \in L_k$ then $\mathbf{b}[\bar{\tau}] \in \mathcal{B}_k$. We also assume that from x the address of the bucket $\mathbf{b}[x]$ can be efficiently computed.

We divide both the buckets and the `tray` into compartments each containing c_2 memory cells, where $c_2 > 0$ is a constant. A compartment is called empty, if its cells contain only 0s. A nonempty compartment in a bucket may contain one of the following two types of information about the corresponding state of $\mathcal{T}$:

(i) the evaluation pair $\langle a, v \rangle$ is at τ, where $\tau \in T_n$, $a \in \{0, 1, ..., n-1\}$, $v \in [0, \wp]$

(ii) $\mathbf{p}(\tau) = \mathbf{b}[x]$, where $\mathbf{b}[x] \in \mathcal{B}_k$ if $\tau \in L_k$. (This corresponds to the coupon $\mathbf{c}(\tau)$.)

A nonempty compartment in the tray may contain one of the following two types of information about the corresponding state of $\mathcal{T}$:

(iii) the evaluation pair $\langle a, v \rangle$ is in the `tray`, where $\tau \in T_n$, $a \in \{0, 1, ..., n-1\}$, $v \in [0, \wp]$

(iv) $\mathbf{p}(\tau) = \mathbf{b}[x]$ is in the tray, where $\mathbf{b}[x] \in \mathcal{B}_k$ if $\tau \in L_k$.

The information of

type (i) will be represented by the quadruplet $\langle 1, a, v, \bar{\tau} \rangle$,

type (ii) by the triplet $\langle 2, \bar{\tau}, x \rangle$,

type (iii) by the quadruplet $\langle 3, a, v, 0 \rangle$,

type (iv) by the triplet $\langle 4, \bar{\tau}, x \rangle$.

The set of all quadruplets and triplets of this type will be denoted by $\mathbf{Q}$. The quadruplets in $\mathbf{Q}$ will be called evaluation-quadruplets, and the triplets in $\mathbf{Q}$ will be called coupon-triplets. These expressions refer to the role of the corresponding notions in the algorithm $\mathcal{T}$.

At the beginning of each time interval $\mathbf{A}_{i,k}$, and $\mathbf{B}_i$ we consider the state of the machine $\mathcal{R}$, where the algorithm $\mathcal{T}$ is running. We assume that $\mathcal{T}$ is running on $\mathcal{R}$ with the same input sequence that is received by $\mathcal{A}'$ on $\mathcal{M}_q$. At the beginning of these intervals the state of the machine $\mathcal{R}$ determines the contents of the buckets and the `tray` in the corresponding state of $\mathcal{A}'$ in the following way. What we describe below will determine only the set of elements of $\mathbf{Q}$ that is contained in the compartments of a bucket or the tray. The same set naturally can be realized in many different ways, by putting the same elements in the compartments in different orders. We will also assume that one element of Q is contained only in a single compartment of a bucket or the tray, and a compartment contains at most one such element.

The following rules define the contents of the buckets and the `tray`.

(a) if an evaluation pair $\langle a, v\rangle$ is at node τ then a compartment of the bucket $\mathbf{p}(\tau)$ contains the evaluation-quadruplet $\langle 1, a, v, \bar{\tau}\rangle$,

(b) if a coupon $\mathbf{c}(\tau)$ is at node σ then a compartment of the bucket $\mathbf{p}(\sigma)$ contains the coupon-triplet $\langle 2, \bar{\tau}, x\rangle$, where $\mathbf{p}(\tau) = \mathbf{b}[x]$,

(c) if an evaluation pair $\langle a, v\rangle$ is in the tray, then a compartment of `tray` contains the eavaluation-quadruplet $\langle 3, a, v, 0\rangle$

(d) if a coupon $\mathbf{c}(\tau)$ is in the tray, then a compartment of the `tray` contains the coupon-triplet $\langle 4, \bar{\tau}, x\rangle$ where $\mathbf{p}(\tau) = \mathbf{b}[x]$,

(e) every compartment is empty unless it contains an element of $\mathbf{Q}$ according to rules (a),(b),(c), or (d),

(f) there are no two different compartments of the same bucket or the tray containing identical elements of $\mathbf{Q}$, and each compartment contains at most one element of $\mathbf{Q}$

Assume that $\mathcal{A}'$ is able to maintain the contents of the buckets and the tray as described. Then $\mathcal{A}'$ can give the correct output in interval $\mathbf{A}_{i,d+2}$ the same way as it was given by $\mathcal{T}$. Indeed, the information what is needed for the correct output is in $\mathbf{b}_{0,0}$. (We assume that $\mathcal{A}'$ always keeps the last input somewhere in the first $\tilde{\gamma}$ memory cells.)

We have to show that $\mathcal{A}'$ is able to update the contents of the buckets and the tray as they are determined by rules (a)-(f).

Below we describe what $\mathcal{A}$ is doing in each interval $\mathbf{A}_{i,k}$. $\mathbf{A}_{i,k}$ will be also an element of the sequence of intervals $J_0, J_1, ...,$ say, $\mathbf{A}_{i,k} = J_{h(i,k)}$. When we define the action of $\mathcal{A}'$ in $\mathbf{A}_{i,k} = J_{h(i,k)}$ we will also tell what will be the set of memory cells $M_{h(i,k)}$, that $\mathcal{A}'$ is using in this interval apart from the first $\tilde{\gamma}$ memory cells.

The defintion of $\mathcal{A}'$ in the intervals $\mathbf{A}_{i,0}$, $i \geq 1$. Here only the tray and 1_{T_n} are used by $\mathcal{T}$. Therefore $\mathcal{A}'$ can execute the necessary changes in the memory of $\mathcal{M}_q$ in poly$(\log n)$ time with $M_{h(i,0)} = \mathtt{tray} \cup \mathbf{b}_{0,0}$.

The definition of $\mathcal{A}'$ in the intervals $\mathbf{A}_{i,s}$, $i \geq 1$, $s = 1, ..., d$. There are two cases. In both cases $M_{h(i,s)} = \mathtt{tray} \cup \mathbf{B}_{s,\eta_{i,s}}$

Case I. The coupon $\mathbf{c}(\tau)$ is not in the tray. In this case $\mathcal{A}'$ does not do anything. (Of course this will not be true for $\mathcal{A}$)

Case II. The coupon $\mathbf{c}(\tau)$ is in the tray. This means that the coupon-triplet $\langle 4, \bar{\tau}, x\rangle$ is in the tray, where $\mathbf{p}(\tau) = \mathbf{b}[x] = \mathbf{b}_{s,j}$ and $\eta_{i,s} = \zeta_{i,\tau}$. This that implies $\mathcal{A}'$ knows $\mathbf{p}(\tau) \subseteq M_{h(i,s)}$ and so $\mathcal{A}'$ can execute the required changes in poly$(\log n)$ time.

In the last two intervals there is nothing problematic.

The definition of $\mathcal{A}'$ in the intervals $\mathbf{A}_{i,d+1}$. $M_{h(i,d+1)} = \mathbf{b}_{0,0} \cup \mathtt{tray}$. $\mathcal{A}'$ puts everything from the tray into $\mathbf{b}_{0,0}$.

The definition of $\mathcal{A}'$ in the intervals $\mathbf{A}_{i,d+2}$, $M_{h(i,d+2)} = \mathbf{b}_{0,0}$. Here $\mathcal{A}'$ gives the same output as $\mathcal{T}$.

Clearly the sets $M_{h(i,s)}$, defined above, have all of the properties that have been expected of them, in particular they can be defined from the random variables $\eta_{i,s,\mathcal{I}}$.

The definition of $\mathcal{A}'$ in the intervals $\mathbf{B}_i$, $i \geq 1$. To reflect the action of $\mathcal{T}$ in the corresponding time interval, the contents of the buckets in the buffers $\mathcal{B}_0, ..., \mathcal{B}_{\kappa(i)}$ has to be changed, but the contents of all of the other buckets will remain unchanged. $\bigcup_{j=0}^{\kappa(i)} \mathcal{B}_j$ will be denoted by $\mathbf{K}$.

Motivation. In this time interval the new value of $\mathbf{p}(\tau)$ must be randomized for all $\tau \in \mathbf{L}_{\kappa(i)}$. Recall that $\mathbf{p}(\tau)$ is a bucket where, e.g., the evaluation quadruplet $\langle 1, a, v, \bar{\tau} \rangle$ must be kept, whose meaning is that in the algorithm $\mathcal{T}$ the pair $\langle a, v \rangle$ is at node τ.

For the moment let us consider only the problem of randomization of the new value of $\mathbf{p}(\tau)$ and the movements of these quadruplets to their new locations. Form this point of view only, the simplest solution would be for $\mathcal{A}'$, to go along the buckets in $\mathcal{B}_0, ..., \mathcal{B}_{\kappa(i)}$ and if in a bucket $\mathbf{b}$ a quadruplet $\langle 1, a, v, \bar{\tau} \rangle$ is found, to randomize immediately the new value $\mathbf{b}'$ of $\mathbf{p}(\tau)$ and to write it into an empty cell of all of the compartments of the bucket, where there is an element of $\mathbf{Q}$ which has to go to $\mathbf{b}'$. Then later, when this has been done for all of the buckets in $\mathcal{B}_0, ..., \mathcal{B}_{\kappa(i)}$ the quadruplets can be taken obliviously to their new destinations determined by the new value of $\mathbf{p}$. An oblivious algorithm which does this step efficiently, the oblivious hashing, was introduced by Ostrovsky, see [9] or [7], and we will use it in the form that will be described later in Lemma 11. The problem of with this solution is that not all of the elements of $\mathbf{Q}$ that must be taken into bucket $\mathbf{b}'$ are in bucket $\mathbf{b}$ at the time of the randomization. For example there may be coupon triplets that has to go there but a the moment they are not in $\mathbf{b}$.

The solution for this problem is the following. Our first goal is to take everything whose final destination is the new bucket $\mathbf{p}(\tau)$ (which has not been randomized yet), to the bucket $\mathbf{b}[\bar{\tau}]$. For this step we use the mentioned oblivious hashing. This is possible since the function $\mathbf{b}[\bar{\tau}]$ can be computed efficiently. Consequently $\mathcal{A}'$ may write the address of $\mathbf{b}[\bar{\tau}]$ into each compartment, where there is an element of $\mathbf{Q}$, which has to go into $\mathbf{b}[\bar{\tau}]$ in this first step. This address written into the compartments will be denoted by $\texttt{destination}_1$. (Step (39) below.)

Then, when everything, whose final destination is in the new bucket $\mathbf{p}(\tau)$, is in $\mathbf{b}[\tau]$, the algorithm $\mathcal{A}'$ goes over all the buckets in $\mathcal{B}_0, ..., \mathcal{B}_{\kappa(i)}$, and at bucket $\mathbf{b}[\tau]$ it randomizes the new value of $\mathbf{p}(\tau)$ and writes its address into all of the compartments where it is needed. (This address will be denoted by $\texttt{destination}_2$, see step (43) below.) Then, with another oblivious hashing everything gets into its final place.

There is an additional complication. Namely, when a coupon $\mathbf{c}(\tau)$ is activated, then for the corresponding coupon triplet $\langle 2, \bar{\tau}, \bar{\mathbf{p}}(\tau) \rangle$ must be put into the bucket $\mathbf{p}(\tau^\circ)$. (Here at both places the newly randomized values of the function $\mathbf{p}$ must be taken.) Therefore, for the proper creation and placement of this coupon triplet, $\mathcal{A}'$ must know the new value of $\mathbf{p}$ both at τ and τ° at the same time. This, however, will not cause a real problem, since the new $\mathbf{p}(\tau)$ is randomized in $\mathbf{b}[\bar{\tau}]$ and the new $\mathbf{p}(\tau^\circ)$ is randomized in $\mathbf{b}[\bar{\tau^\circ}]$. Therefore, if $\mathcal{A}'$ once goes over all of the pairs $\mathbf{b}(\bar{\tau})$, $\mathbf{b}[\bar{\tau^\circ}]$, it may transfer the needed information into the buckets where the coupon triplets will be created. (Step (42) below.) *End of motivation.*

We divide the action of $\mathcal{A}'$ into the following phases

(38) *$\mathcal{A}'$ goes along all of the buckets in $\mathbf{K}$, erases all of the coupon-triplets of the form $\langle 2, \bar{\tau}, x \rangle$, where $\tau \in \mathbf{L}_{\kappa(i)}$. (This correspond to step* (15) *of $\mathcal{T}$, that is, the destruction of certain coupons.)*

In case of step (38) the intervals J_j will be the intervals why while $\mathcal{A}$ is working on a single bucket B. For such an interval, $M_j = B$. If $\mathcal{A}$ chooses the buckets in order of their addresses, clearly the sequence M_j can be efficiently computed using only i, which can be stored in one of the first $\tilde{\gamma}$ cells.

(39) *$\mathcal{A}'$ goes along all of the buckets in $\mathbf{K}$. Suppose that in a compartment C, it finds a coupon-triplet $\langle 2, \bar{\tau}, x \rangle$, where $\tau \in T_n \backslash \mathbf{L}_\kappa$. Let σ be the unique element of $L_{\kappa(i)}$ with $\sigma \geq \tau$. Assume now that $\mathcal{A}'$ finds an evaluation quadruplet $\langle 1, a, v, \bar{\tau} \rangle$. In this case let σ be the unique element of $L_{\kappa(i)}$, so that $\sigma \in \mathtt{branch}(\chi(a))$. In both cases $\mathcal{A}'$ writes $\bar{\sigma}$ into compartment C, leaving there also the element of $\mathbf{Q}$ which was contained in C at the beginning of this step. $\bar{\sigma}$ will be denoted by $\mathtt{destination}_1(C)$.*

The intervals J_j are defined in the same way as in step (38). The set M_j is the union of the bucket in question, and the set $M^{(\chi)}$ where the poly$(\log n)$ random bits defining the function χ are stored.

(40) *For each compartment C in the buckets of $\mathbf{K}$, $\mathcal{A}'$ does the following. Suppose that $\mathtt{destination}_1(C) = \bar{\tau}$. (By the definition of the function $\mathtt{destination}_1$ we have $\tau \in L_{\kappa(i)}$.) Then $\mathcal{A}'$ moves the contents of compartment C into an empty compartment C' of bucket $\mathbf{b}[\bar{\tau}]$, then erases from C' the integer $\mathtt{destination}_1(C)$, leaving there only a single element of $\mathbf{Q}$. $\mathcal{A}'$ does this in a way that each compartment will contain at most one element of $\mathbf{Q}$. (We will describe later the way this step can be accomplished by $\mathcal{A}'$.)*

(41) *$\mathcal{A}'$ goes over all of the of buckets in $\mathbf{K}$ and for each $\tau \in \mathbf{L}_\kappa$, $\tau \in L_s$ it takes a random value of $\zeta_{i,\tau}$. (The new value of $\mathbf{p}(\tau)$ is the bucket $\mathbf{b}_{s,\zeta_{i,\tau}}$). Suppose that $\mathbf{b}_{s,\zeta_{i,\tau}} = \mathbf{b}[x]$. $\mathcal{A}'$ writes x into an empty compartment of $\mathbf{b}[\bar{\tau}]$. We will denote this compartment by $C_{0,\tau}$*

The intervals J_j and the sets M_j are defined as in step (38)

(42) *$\mathcal{A}'$ goes over all of the pairs of buckets $\mathbf{b}(\bar{\tau})$, $\mathbf{b}[\bar{\tau^{\circ}}]$ in $\mathbf{K}$ and for each $\tau \in \mathbf{L}_\kappa$, it writes in an empty compartment C of $\mathbf{b}[\bar{\tau^{\circ}}]$ the coupon-triplet $\langle 2, \bar{\tau}, x \rangle$, where x is the content of compartment $C_{0,\tau}$, which is in $\mathbf{b}[\bar{\tau}]$.*

The intervals J_j are the time intervals while a single $\tau \in \mathbf{L}_\kappa$ is handled. The corresponding M_j is $\mathbf{b}[\bar{\tau}] \cup \mathbf{b}[\bar{\tau^{\circ}}]$.

(43) *$\mathcal{A}'$ goes over all of the buckets in $\mathbf{K}$ and at bucket $\mathbf{b}[\bar{\tau}]$, at each compartment C which contains an element of $\mathbf{Q}$ it writes into an empty memory cell of C the content of compartment $C_{0,\tau}$. The content of this cell will be denoted by $\mathtt{destination}_2(C)$. After that it erases the content of $C_{0,\tau}$.*

The time intervals J_j are the intervals while a single bucket is handled and the set M_j is the bucket itself.

(44) *For each compartment C in the buckets of $\mathbf{K}$, $\mathcal{A}'$ does the following. Suppose that $\mathtt{destination}_2(C) = \bar{\tau}$. (By the definition of the function $\mathtt{destination}_2$ we have $\tau \in \mathbf{L}_{\kappa(i)}$.) $\mathcal{A}'$ moves the content compartment C into an empty compartment C' of bucket $\mathbf{b}[\bar{\tau}]$, then erases from C' the integer $\mathtt{destination}_2(C)$, leaving in C' only a single element of $\mathbf{Q}$. $\mathcal{A}'$ does this in a way that each compartment will contain at most one element of $\mathbf{Q}$. (This step will be accomplished in the same way as step (44).)*

This, apart from the details of step (40) and (44) completes the definition of $\mathcal{A}'$.

Both in step (40) and step (44) we have to move the contents of compartments into destination buckets, given by the functions $\mathtt{destination}_1$ and $\mathtt{destination}_2$. To do this we will use Ostrovsky's "oblivious hashing" method see [8], [9]. A slightly different version of the oblivious hashing is given in [7]. Either one is suitable to prove the following lemma which can be used in our case.

In the following definitions and in Lemma 11 below the words compartment and bucket will be used only for blocks of memories without their special structure that we have used till now. We will apply Lemma 11 however for this specific meaning of the words buckets and compartments

Definition. 1. Assume that a part of the memory of $\mathcal{M}_q$ is divided into α consecutive blocks $B_0, ..., B_{\alpha-1}$ called buckets and each bucket, is partitioned into β consecutive blocks called compartments. Each compartment contains γ memory cells. The compartments in their natural order, according their addresses will be denoted by $C_0, ..., C_{\alpha\beta-1}$

Assume that an algorithm $\mathcal{G}$ changes the contents of the memory cells in the buckets. The algorithm starts at time 0 and stops at time $\bar{t}$. At time t the content of compartment C_i will be denoted by $\mathtt{cont}_t(C_i)$ for $i = 0, 1, ..., m-1$. The content of the first memory cell in C_i at time 0 will be denoted by $\mathtt{destination}(C_i)$. We assume that $\mathtt{destination}(C_i) \in \{0, 1, ..., \alpha-1\} \cup \{\infty\}$, for all $i = 0, 1, ..., \alpha\beta - 1$. The element ∞ can be encoded e.g., by $2^q - 1$. (Therefore if $\mathtt{destination}(C_i) \neq \infty$ we may think of $\mathtt{destination}(C_i)$ as one of the buckets $B_0, ..., B_{\alpha-1}$). Let $H = \{i \in \alpha\beta \mid \mathtt{destination}(C_i) \neq \infty\}$.

We say that the algorithm $\mathcal{G}$ moves the contents of the compartments to their destinations, if there exists a one-to-one map δ of H into $\alpha\beta$ so that the following conditions are satisfied

(45) *for each $i \in H$, compartment $C_{\delta(i)}$ is in bucket $B_{\mathtt{destination}(C_i)}$ and* $\mathtt{cont}_{\bar{t}}(C_{\delta(i)}) = \mathtt{cont}_0(C_i)$

(46) *for all $j \in \alpha\beta$ if there is no $i \in H$ with $j = \delta(i)$, then* $\mathtt{cont}_{\bar{t}}(C_j) = 0$.

2. We say that the function $\mathtt{destination}$ is contradictory if there exists a bucket B_j, $j \in \{0, 1, ..., \alpha-1\}$ so that the number of all compartments C_i, $i \in \alpha\beta$, with $j = \mathtt{destination}(C_i)$ is more than β, the number of compartments in B_j.

Remark. According to this definition, $\mathcal{G}$ moves the contents of the compartments to their destinations, if the content of each compartment is moved into a bucket, which is designated by the function `destination`, at least for those buckets where such a destination is given, that is, the value of the function is not ∞. The function `destination` can be defined in a way that such an algorithm $\mathcal{G}$ cannot exist. Namely, this is the case, if the contents of more compartments should be taken into a single bucket than its capacity. In this case we say that the function `destination` is contradictory. Obviously, if it is not contradictory, then there is an algorithm $\mathcal{G}$ which moves the contents of the compartments to their destinations. The next lemma says that there is such a deterministic algorithm $\mathcal{G}$, which is also oblivious and efficient.

Lemma 11 *There exists a $c > 0$ and a deterministic and oblivious algorithm $\mathcal{G}$, so that for all q, if we fix the values of the positive integers α, β, γ arbitrarily with $\alpha\beta\gamma < 2^{q-3}$, and if the* *contents of the compartments* $C_0, ..., C_{\alpha\beta-1}$ *are given in a way that the function* `destination` *is not contradictory, then, in time* $c\gamma\alpha\beta\log(\alpha\beta)$ *and using no more than* $c\alpha\beta\gamma$ *memory cells,* $\mathcal{G}$ *moves the contents of the compartments* C_i *into their destinations.*

The algorithm $\mathcal{G}$ in the lemma can be constructed and it correctness proved in the same way, using oblivious sorting networks, as the oblivious hashing is done in [9] or [7]. To make the paper more selfcontained we give a proof of lemma 11 following the ideas of the corresponding proof in [7].

Proof of Lemma 11. We may assume that for some constant c_2 whose value will be fixed later, the the content of the last c_2 memory cells of each compartment is 0. Indeed, if this does not hold, $\mathcal{G}$ can copy everything into new buckets and compartments with larger sizes, and at the end may copy back the result into the original compartments. All of this takes only space and time linear in $\alpha\beta\gamma$. The importance of this assumption is that $\mathcal{G}$ may strore extra information in the last c_2 cells of each compartment. The set of last c_2 cells of compartment C_i will be denoted by D_i.

According to our definition, we have α buckets $B_0, ..., B_{\alpha-1}$. We create 3α new buckets $B_\alpha, B_{\alpha+1}, ..., B_{4\alpha-1}$ of the same sizes as the original ones, that is, each new bucket ontains β compartments, each consisting of γ cells. Now we have altogether 4α buckets $B_0, ..., B_{4\alpha-1}$. The new compartments will be denoted by $C_{\alpha\beta}, C_{\alpha\beta+1}, ..., C_{4\alpha\beta-1}$, where compartments $C_{k\beta,...,(k+1)\beta-1}$ form the new bucket B_k, for $k = \alpha, ..., 4\alpha - 1$. Our first goal is to show that there exists an algorithm $\mathcal{G}'$, that may set the contents of the new $3n$ buckets, in an oblivious way, and within the given bounds on space and time, so that the contents established this way, satisfy the following conditions.

(47) *For each* $i = \alpha, ..., 4\alpha - 1$*, the content of the first cell of* D_i *is* 1*. (This distinguishes the contents of the new compartments from the old ones.)*

(48) *Assume that we extend the function* `destination` *to the new compartments so that the content of the first memory cell of* C_i *is* `destination`(C_i) *for all* $i = \alpha, ..., 4\alpha - 1$*. Then,* `destination`$(C_i) \in \{0, 1, ..., \alpha - 1\} \cup \{\infty\}$ *for all* $j = 0, 1, ..., 4\alpha - 1$*. Morevover for each* $j = 0, 1, ..., \alpha - 1$ *there exists exactly* β *elements* i *of the set* $\{0, 1, ..., 4\alpha - 1\}$ *so that* `destination`$(C_i) = j$

First we show that the existence of a deterministic algorithm $\mathcal{G}'$ with the described conditions implies the lemma. Assume that after the work of $\mathcal{G}'$ the contents of the compartments satisfy conditions (47) and (48).

After that $\mathcal{G}$ moves the contents of some compartments into other compartments so that at the end the contents of the compartments are sorted according to the sizes of their destinations. More precisely assume that at time t' conditions (47) and (48) are satisfied. $\mathcal{G}$ works till time t'' so that, for a suitably chosen a permutation π of the set $\{0, 1, ..., 4\alpha - 1\}$ the following holds.

(49) *for all* $i \in 4\alpha$*,* $\mathtt{cont}_{t''}(C(i)) = \mathtt{cont}_{t'}(C_{\pi(i)})$

(50) *for all* $i, j \in 4\alpha$*,* $i \le j$ *implies* $\mathtt{destination}_{t''}(C_i) \le \mathtt{destination}_{t''}(C_j)$

This sorting is done by $\mathcal{G}$ using an oblivious sorting network, e.g., Batcher's algorithm, (see [4]), or the AKS network, see [2]. With the latter one, we get a time upser bound $O(\alpha\beta\gamma(\log\alpha\beta))$. With Batcher's algorithm we will have only an $O(\alpha\beta\gamma(\log\alpha\beta))^2$, time upper bound but for practical values of α, β and γ this solution may be faster. The sorting network is used so that at each step the contents of two predetermined compartments are either swapped or not depending on the result of the comparison of their destinations. The expression "predetermined" means that the sequence of pairs of compartments used in these comparisons depend only on α, β, and γ. At the end, the contents are in the compartments, sorted according to the values of the function `destination`. Condition (48) implies that each content got into the bucket determined by its destination. After that $\mathcal{G}$ goes over all of the compartments in their natural order and puts zeros where the content of D_i indicates that it originated in a new compartment. This completes the proof of the lemma if we accept the existence of $\mathcal{G}'$ with the described properties.

We show now that there exists an algorithm $\mathcal{G}'$ with the required properties. For each $j = 0, 1, ..., \alpha - 1$, we will denote by $h_t(j)$ the total number of compartments C_i (both old and new) with $\texttt{destination}_t(C_i) = j$. $\mathcal{G}'$ starts to work at time 0. the assumption that the function `destination` is not contradictory implies that $h_0(j) \in [0, \beta]$ for all $j \in \alpha$. We define $\mathcal{G}'$ as follows.

Phase I, in time interval $[0, t_1]$ $\mathcal{G}'$ goes over the compartments $C_{\alpha\beta}, C_{\alpha\beta+1}, ..., C_{2\alpha\beta-1}$ and, for each $i = \alpha\beta, ..., 2\alpha\beta - 1$, changes the content of the first cell in C_i so that we get $\texttt{destination}_{t_1}(C_i) = \lfloor \frac{i-\alpha\beta}{\beta} \rfloor$. *End of Phase I.*

The action of $\mathcal{G}'$ in Phase I implies that for all $j = 0, 1, ..., \alpha - 1$ we have $h_{t_1}(j) = h_0(j) + \beta$ and $\beta \le h_{t_1}(j) \le 2\beta$. The reason is, that among $C_0, ..., C_{\alpha\beta-1}$ there exist at most β compartments C_i with $\texttt{destination}_{t_1}(C_i) = j$, and among $C_{\alpha\beta}, ..., C_{2\alpha\beta-1}$ there exists exactly β such compartments. For all $i = 2\alpha\beta, ..., 4\alpha\beta - 1$ we have $\texttt{destination}_{t_1}(C_i) = \infty$.

Phase II, time interval $(t_1, t_2]$. $\mathcal{G}'$ sorts the contents of the compartments, according to the values of the function `destination`, using the sorting techniques described earlier. *End of Phase II.*

Clearly we have $h_{t_2}(j) = h(t_1)(j) = h_0(j) + \beta$ for all $j \in \alpha$.

Phase III, time interval $(t_2, t_3]$. $\mathcal{G}'$ goes along the compartments $C_0, ..., C_{2\alpha\beta-1}$ in this order and for each $j = 0, 1, ..., \alpha - 1$ it counts how many compartments C_i exist with $\texttt{destination}(C_i) = j$. (That is, it determines the determines the current value of value $h(j)$. Since the value of h will not change during this phase, $h(j)$ always the same as $h_2(j)$.) Because of the sorting performed by $\mathcal{G}'$ in Phase II, we have that for a fixed $j \in \alpha$, all of the compartments C_i with $\texttt{destination}(C_i = j)$ form a block of consecutive compartments. The length of this block is in the interval $[\beta, 2\beta]$. When $\mathcal{G}'$ realizes that a block has ended then it stores the pair $\langle j, h(j)\rangle$. For the storage of these pairs $\mathcal{G}'$ is using a set Y of memory cells, which do not belong to any of the buckets or compartment. Y contains only a constant number of memory cells. As we describe below, pairs of the type $\langle j, h(j)\rangle$ will be regularly deleted from Y, to make space for other pairs.

When $\mathcal{G}'$ is at compartment $C_{k\beta}$ for some $k = 1, ..., \alpha$ it checks whether there is a pair $\langle j, h(j)\rangle$ in Y, and if the answer is yes, then $\mathcal{G}'$ writes one of them, say, the pair of $\langle j, h(j)\rangle$ into compartment $C_{\alpha\beta+2k\beta}$ using only cells of $D_{\alpha\beta+2k\beta}$ which has not been used yet for any other purpose. After that it erases the pair $\langle j, h(j)\rangle$ from Y. The bound "$\beta \le h(j)$ for all $j \in \alpha$", implies that no more than one pair $\langle j, h(j)\rangle$ will be in Y at any time, and so after $\mathcal{G}'$ performed the described step for $k = \alpha$ the set Y is empty, and for each $j = 0, 1, ..., \alpha - 1$ the pair $\langle j, h(j) \rangle$ was written into D_i, for some $i \in \{2\alpha\beta, ..., 4\alpha\beta - 1\}$. *End of Phase III*

Note that $h_{t_3}(j) = h_{t_2}(j) = h_0(j) + \beta$ for all $j \in \alpha$.

Phase IV, time interval $(t_3, t_4]$ $\mathcal{G}'$ goes over tha compartments $C_{4\alpha\beta-1}, ..., C_{2\alpha\beta}$ in this order. Assume that in compartment $2k\beta - 1$ it finds the pairs $\langle j, h_{t_2}(j) \rangle$, where $j \in \alpha$. Then $\mathcal{G}'$ changes the content of the first memory cells of compartments C_r into j for $r = 2k\beta - 1, ..., 2k\beta - (2\beta - h_{t_2}(j))$. The result of this is that among the compartments $C_{2k\beta-1}, ..., C_{2(k-1)\beta}$ there will be exactly $2\beta - h_{t_2}(j)$ compartment C_r with $\mathtt{destination}_{t_4}(C_r) = j$. (This step can be performed since $h_{t_2}(j) \in [\beta, 2\beta]$ implies that $2\beta - h_{t_2}(j) \in [0, \beta]$. *End of Phase IV*

Note that $h_{t_4}(j) = h_{t_3}(j) + 2\beta - h_{t_2}(j) = h_{t_2}(j) + 2\beta - h_{t_2}(j) = 2\beta$

Phase V, time interval $(t_4, t_5]$. As a next step $\mathcal{G}'$ changes the first cells, that is, the value of $\mathtt{destination}(C_i)$ into ∞, for all $i = 0, ..., 4\alpha\beta - 1$, where the content of first element of D_i indicates that it originated in a new compartment. *End of Phase V.*

Since we have created exactly β new compartment with $\mathtt{destination}(C_i) = j$, we have $h_{t_5}(j) = h_{t_4}(j) - \beta = \beta$ for all $j \in \alpha$ as required in the definition of

We claim that when all of these steps are completed at time t_5, we will have $h_{t_5}(j) = \beta$ for all $j = 0, 1, ..., \alpha - 1$ as required in the definition of $\mathcal{G}'$. *Q.E.D.*(Lemma 11)

Using this lemma we get that both step (40) and step (44) can be performed by a deterministic and oblivious algorithm in poly$(\log n)2^{\kappa(i)}$ times. Since $\mathcal{A}'$ is oblivious and deterministic in this interval, the time intervals J_j and the sets M_j can be trivially constructed, say each instruction of $\mathcal{A}'$ can be an interval J_j. This completes the definition of the algorithm $\mathcal{A}$.

5 Simulating a protected CPU

Definition. 1. We assume that the machine $\mathcal{M}_q$ interpretes the q bits contained in a single memory cell as a signed integer in the interval $[-2^{q-1} + 1, 2^{q-1} - 1]$. (An alternative assumption could be that the q bit is interpreted as a nonnegativ integer in the interval $[0, 2^q - 1]$. Everything that we prove in this section remains true for this situation as well, in fact the proof of the analogue Lemma 13 is easier in that case.)

2. If q is a positive integer, then the set of all integers in the interval $[-2^{q-1} + 1, 2^{q-1} - 1]$ will be denoted by U_q. For each q the following functions, as computed by the corresponding intructions of $\mathcal{M}_q$, will be called *basic q-arithmetic functions*: the binary functions $x + y$, $x \times y$, $x - y$, $\lfloor x/y \rfloor$ as binary and the constants (0-ary functions) $0, 1$, and $2^{q-1} - 1$. The 0-ary function $2^{q-1} - 1$ will be also denoted by $\mathbf{c}_q$.

Remark. 1. All of these functions can be computed by the machine $\mathcal{M}_q$ with a single instruction. We may add a constant number of other arbitrary other functions $f(x, y)$, defined on U_q ,to the set of basic q-arithmetic functions, if we assume that the machine $\mathcal{M}_q$ has an arithmetic instruction that computes $f(x, y)$. Everything that we prove in this section remains true for this extended notion of q-arithmetic functions and for the corresponding machines $\mathcal{M}_q$. Theorem 1 remain valid for this modified machine $\mathcal{M}_q$. For example the program P can use the instruction $\sqrt{x}$, which gives an approximate value of the square root of x. In this case program P_1 of Theorem 1 can also use this instruction.

2. The assumption that the constant $2^{q-1}-1$ is included as an arithmetic instruction (a built in constant) of $\mathcal{M}_q$ may seem arbitrary. If there is no such a built in constant, it is equaly good if there are other built in constants so that $2^{q-1}-1$ can be computed from them with an arithmetic expression of constant depth. If even this is not possible, then, when the machine starts to work we may compute first $2^{q-1}-1$ and store it in a memory cell. This way, we will be able to simulate the instruction which gives the value $2^{q-1}-1$ with a read instruction.

3. The arithmetic operations $x+y, xy, \lfloor x/y \rfloor$, as defined in mathematics, cannot be always performed on numbers given by at most q bits so that the result is well-defined and it also has at most q bits. In such a case we say an "overflow" occured. We do not define how $\mathcal{M}_q$ handles the overflows, our algorithm in the proof of Lemma 2 has the property that overflow can never occurs, However, the lemmas below are easier to formulate if we assume that the machine $\mathcal{M}_q$ always provides a result for the arithmetic operations $x+y, xy, \lfloor x/y \rfloor$. The lemmas below, in particular Lemma 13, hold independently of the definitions of these values.

Definition. 1. If we do not fix a value of q then $x+y$, $x \times y$, $x-y$, $\lfloor x/y \rfloor$, 0, 1, and $\mathbf{c}$ will denote the families of q-arithmetic functions, which has a unique element for each positive integer q. $\mathcal{A}$ will denote the set of these families that is $\mathcal{A} = \{x+y, x \times y, x-y, \lfloor x/y \rfloor, 0, 1, \mathbf{c}\}$.

2. An arithmetic circuit C is a quadruplet $C = \langle G, F, I \leq_G \rangle$ with the following properties:

$G = \langle V, E \rangle$ is a finite directed graph wihtout cycles,

F is a function defined on V with values in $\mathcal{A}$,

I is a subset of V, so that each element of I has indegree 0.

$\leq_G$ is an ordering of the set V,

For each $x \in V \backslash I$ the indegree of x is equal to the arity of $F(x)$.

The elements of I will be called the input nodes and the elements with outdegree 0 will be called the output nodes of the circuit.

The depth of an element x is the maximal length taken for all of the directed paths which end in x. This implies that the depth of all input nodes are 0. There may be other nodes x whose depth is 0. For such a node x, $F(x)$ is a 0-ary family of functions, e.g. the constant 1.

3. Suppose that C is an arithmetic circuit and q is a positive integer. If C has exactly k input nodes $x_1 < ... < x_k$ and it has exactly l output nodes $y_1 < ... < y_l$ the we define a k-ary function $f_{C,q}$ on U_q with values in U_q^l. Assume that $a_1, ..., a_k \in U_q$. $f_{C,q}(a_1, ..., a_k)$ is defined as follows. By recursion on the depth of the element $v \in V$ we attach to each element of v an element g_v depending on $v_1, ..., v_k$. If v is an input node then $v = x_j$ for exactly one $j = 1, ..., k$ and then by definition $g_v = a_j$. If v is a node of depth 0 which is not an input node, then $g_q(v)$ is the value of the constant $F(v)$ in U_q. Assume now that the depth of v is $i > 0$ and g_u has been already defined for each $u \in v$ whose depth is less than i. Suppose that the indegree of v is j and $u_1 <_G < ... <_G u_j$ are all of the vertices that has an outgoing edge pointing to v. Then $g_v = (F(v))(g_{u_1}, ..., g_{u_j})$. Finally $f_{C,q} = \langle g_{y_1}, ..., g_{y_l} \rangle$.

4. The previous definition gives for each positive integer q a function $f_{C,q}$. The family of these functions for $q = 1, 2, ...$ will be denoted by f_C. If a family of functions for $q = 1, 2, ...$ is of the form f_C for a suitably chosen arithmetic circuit C, then we will say that the family of functions f_C is arithmetic.

5. Assume that $f^{(q)}(x_1, ..., x_k)$, $q = 1, 2, ...$ is an arithmetic family of functions and for all $i = 1, ...k$, $g_i^{(q)}(z_{i,1}, ..., z_{i,j_i})$, $q = 1, 2, ...$ is also and arithmetic family of functions. Then

the composition of these families defined as the family $h^{(q)}(z_{1,1},...,z_{1,j_1},...,z_{k,1},...,z_{k,j_k}) = f^{(q)}(g_1^{(q)}(z_{1,1},...,z_{1,j_1}),...,g_k^{(q)}(z_{k,1},...,z_{k,j_k}))$ $q = 1,2,....$

Proposition 8 *The composition of arithmetic families of functions is also arithmetic.*

Proof. The statement of the proposition is an immediate consequence of the fact that we can build a circuit computing the composite function from the circuits computing its constituents. *Q.E.D.*(Proposition PR0)

Proposition 9 *For all $m > 0$ if $P(X_1,...,X_m)$ is a propositional formula containing only the boolean variables $X_1,...,X_m$ (and no other atomic formulas), then then there is an arithmetic family of functions $f^{(q)}(x_1,...,x_m)$ so that for all $\langle x_1,...,x_m\rangle \in \{0,1\}^m$ and for all $q = 1,2,...,$ we have that $f^{(q)}(x_1,...,x_m) \in \{0,1\}$ and $f^{(q)}(x_1,...,x_m) = 1$ iff $P(x_1,...,x_m) = 1$.*

Proof. We need only that the operations $+, -, \times$ are among the basic arithmetic instructions of $\mathcal{M}_q$, moreover, if we consider the boolean values $TRUE$ and $FALSE$ as the integers 1 and 0, then $a \wedge b \equiv ab$, $\neg a \equiv 1 - a$, and $a \vee b = 1 - (1-a)(1-b)$. *Q.E.D.*(Proposition 9)

Lemma 12 *Assume that k is a positive integer. For all $q > \max\{10, \log_2 k\}$, we define a function G_q which maps U_q^k into itself. Assume that $a_0,...,a_{k-1} \in U_q$. We consider the machine $\mathcal{M}_q$ with state S, where the content of $\texttt{cell}(i)$ is a_i for $i = 0,...,k-1$, and the contents of the other cells have arbitrary but fixed values. Strating from state S the machine $\mathcal{M}_q$ executes an instruction. Suppose first that only the cells form the sets $\texttt{cell}(0),...,\texttt{cell}(k-1)$ will be involved in the execution of this instruction, that is, the contents of the other cells neither influence the outcome of the instruction nor will be changed as a result of the instruction, and moreover the instruction is neither an input/output instruction nor a random number generator instruction. In this case we define $G_q(a_0,...,a_{k-1})$ by $G_q(a_0,...,a_{k-1}) = \langle b_0,...,b_{k-1}\rangle$, where b_i is the content of $\texttt{cell}(i)$ after the instruction has been executed. Otherwise $G_q(a_0,...,a_{k-1}) = \langle 0,...,0\rangle$.*

Then, there exists an arithmetic family of functions $G^{(q)}$, $q = 1,2,...,$ so that for all $q > \max\{10, \log_2 k\}$ we have $G_q = G^{(q)}$

First accepting Lemma 12 we prove Lemma 1. Actually the use of Lemma 1 for the proof of Theorem 1 can be avoided by applying directly the techniques used in the proof of Lemma 13. We will tell about this later after the formulation of Lemma 13 below.

Proof of Lemma 1. According to Lemma 12 there is a family of arithmetic circuits $C^{(q)}$, $q = 1,2,...$ of size depending only on α, so that if the input of $C^{(q)}$ is the sequence of contents of the memory cells $\texttt{cell}(0),...,\texttt{cell}(\alpha-1)$ at time t' then the output of $C^{(q)}$ is the sequence of the contents of the same memory cells at time $t'+1$, for each $t' \in [0,t]$, provided that the machine $\mathcal{M}_q$ works as described in condition (1) of Lemma 1. Therefore the program P', by evaluating this arithmetic circuit repeatedly, can determine $\texttt{state}(P,t,0,\alpha-1)$. Indeed, the first input of the circuit is the sequence of the initial contents of the memory cells $\texttt{cell}(0),..,\texttt{cell}(\alpha-1)$ at time 0 as given in the initial sate P. The output will be $\texttt{state}(P,1,0,\alpha-1)$. Using now this output as the input of the circiut P' gets $\texttt{state}(P,2,0,\alpha-1)$, and repeating this process at then at the end it gets $\texttt{state}(P,t,0,\alpha-1)$ as required in the lemma. A single evaluation of the arithmetic circuit C_q can be done in time depending only in α, since the size of the circuit also depended only on α. The output of an arithmetic circuit can be computed obliviously, that is, in a way that the visible history of the computation is fixed. E.g., we order the nodes of the circuit in a way that each edge in the directed graph, from the definition of the circuit, points from a smaller node to a larger one. Then we evaluate the circuit by computing the value corresponding to each node x by an arithmetic operation, using the already known values at the tails of the edges whose head is x. Clearly the visible history of this computation depends only on the circuit and not on the actual inputs. This implies that both conditions (1) and (2) can be satsified by P' at the same time. *Q.E.D.*(Lemma 1)

The following lemma will be needed in the proof of Lemma 12

Lemma 13 *The following family of functions defined on* $U_q = [-2^{q-1}+1, 2^{q-1}-1]$ *are arithmetic.*

(51) $f(X,Y) = 1$ *if* $X = Y$, $f(X,Y) = 0$ *otherwise.*

(52) $f(X,Y) = 1$ *if* $X \leq Y$, $f(X,Y) = 0$ *otherwise.*

Remark. This lemma can be used directly to prove Theorem 1, avoiding the use of both Lemma 1 and Lemma 12. Each time, when the protected CPU is used, in the algorithm provided in the proof of Lemma 2, we may avoid using conditional instructions. E.g., assume that the algorithm has to put the value a into `cell`(s) iff $u \leq v$, where u, v are in given memory cells, then we can do the following. Say, the content of `cell`(s) is b. Using Lemma 13 we compute obliviously $f(u,v)$, where $f(u,v) \in \{0,1\}$ and $f(u,v) = 1$ iff $u \leq v$. Then we compute the value $f(u,v)a + (1 - f(u,v))b$ and put it into `cell`(s). Clearly this sequence of instructions perform the required task. This computation is also oblivious provided that the computation of $f(u,v)$ is oblivious, which follows from Lemma 13. Naturally the same thing can be done with the equality relation or any boolean combinations of the equality and ordering relations. Using this simple solution, we can move the contents of memory cells around obliviously, so that the movements depend on conditions which are expressed in terms of the relation equality and ordering. For example, we may swap or not the contents of two memory cells depending on such a condition. Going through the algorithm it is easy to check that everything whih was done in the protected CPU can be made oblivious this way. Naturally, if, depending on a condition, one of two different computations must be performed, then, to make this oblivious, we always have to perform both computations but using the result of only the required one, which can be done in the described way. This method of making the computation oblivious requires more work in the proof, so this is why we rather use the more general Lemma 1 and Lemma 12. However, the algorithm obtained through the direct use of Lemma 13 is much more efficient, and so it is more suitable for a practical implementation.

Proof. Assume that $x_1, ..., x_c \in U_q$ and R_q is an a-ary relation on U_q where a is a constant, and $B_q \subseteq U_q$. We will say that the family of relations R_q, $q = 1, 2, ...$ is perfect with the condition B_q, if there exists an arithmetic expression Φ (using the basic arithmetic functions) which gives the truth value of R_q on the elements of B_q, for all $q = 1, 2, ...$ If the family of relations R_q is perfect with the condition U_q, then we will say that R_q is perfect. It is a consequence of Proposition 9 that the boolean combinations of a finite number of perfect relations are also perfect.

By definition $\lfloor a \rfloor$ is the largest integer not exceeding a. Therefore if $u \in U_q = [-2^{q-1} + 1, 2^{q-1} - 1]$ then $\alpha_u = \lfloor \frac{u}{2^{q-1}-1} \rfloor$ may take three possible values. For $u \in [-2^{q-1} + 1, -1]$, $\alpha_u = -1$, for $u \in [0, 2^{q-1} - 2]$, $\alpha_u = 0$, and for $u = 2^{q-1} - 1$, $\alpha_u = 1$. Consequently

(53) *$\alpha_u u = |u|$ for all $u \in U_q$*

Let $f(x) = \frac{(x+1)(2-x)}{2}$ and let $\beta_u = f(\alpha_u)$. We have that for $u \in [-2^{q-1} + 1, 2^{q-1} - 1]$, $u \geq 0$ implies $\beta_u = 1$, $u < 0$ implies $\beta_u = 0$, that is,

(54) *β_u is the boolean value of the relation $u \geq 0$.*

Therefore β_{-u} is the boolean value of $u \leq 0$ and $\beta_u \beta_{-u}$ of $u = 0$. Consequently

(55) *all of the unary (family of) relations $u \leq 0$, $u < 0$, $u = 0$, $u > 0$, $u \geq 0$, are perfect.*

Our next goal is to prove that

(56) *the family of relations relation $u \leq v$ is perfect*

We claim that we can express the relation $u \leq v$ as a boolean combination of the perfect relations listed in (55) and the relation $|u| \leq |v|$.

Indeed $u \leq v$ iff at least one of the following three conditions are satisfied
(i) $u \geq 0 \wedge v \geq 0 \wedge |u| \leq |v|$
(ii) $u \leq 0 \wedge v \leq 0 \wedge |v| \leq |u|$
(iii) $u \leq 0 \wedge 0 \leq v$

Therefore, in order to prove that the relation $u \leq v$ is perfect, it is sufficient to prove, that the relation $|u| \leq |v|$ is perfect. (We cannot use $u \leq v$ iff $u - v \leq 0$ since $u - v$ may cause an overflow.)

Assume now that $u, v \in B = [0, 2^{q-1} - 2]$. Then $u < v$ iff $\lfloor \frac{1+u}{1+v} \rfloor < 1$. (For $u = 2^{q-1} - 1$ this would cause overflow.) Consequently the relation $x < y$ is perfect with the condition B and so all of the relations $x < y$, $x \leq y$, $x = y$ are perfect with condition B, since they are boolean combinations of relations which are perfect with condition B. (Namely $x \leq y \equiv \neg(y < x)$, and $x = y \equiv x \leq y \wedge y \leq x$.)

For $u, v \in D = [0, 2^{q-1} - 1]$, we use that $u = 2\lfloor \frac{u}{2} \rfloor + u - 2\lfloor \frac{u}{2} \rfloor$, $v = 2\lfloor \frac{v}{2} \rfloor + v - 2\lfloor \frac{v}{2} \rfloor$, and $u < v$ iff

$$\left\lfloor \frac{u}{2} \right\rfloor < \left\lfloor \frac{v}{2} \right\rfloor \vee \left(\left\lfloor \frac{u}{2} \right\rfloor = \left\lfloor \frac{v}{2} \right\rfloor \ \wedge \ u - 2\left\lfloor \frac{u}{2} \right\rfloor < v - 2\left\lfloor \frac{v}{2} \right\rfloor \right)$$

All of the four integers $\lfloor \frac{u}{2} \rfloor$, $u - 2\lfloor \frac{u}{2} \rfloor$, $\lfloor \frac{v}{2} \rfloor$, $v - 2\lfloor \frac{v}{2} \rfloor$ occurring in the comparisons are in the interval $B = [0, 2^{q-1} - 2]$, therefore using the already proven fact that $x < y$ and $x = y$ are perfect with condition B we get that $u < v$ is perfect with condition D. By (53) this implies that $|u| < |v|$ is perfect and so $|u| \leq |v| \equiv \neg|v| < |u|$ is also perfect. Therefore we have proved that the family of relations $u \leq v$, $u, v \in U_q$, $q = 1, 2, ...$ is perfect which implies condition (52). Condition (51) is an immediate consequence of condtion (52) and Proposition 9. *Q.E.D.*(Lemma 13)

Remark. 1. The lemma also holds with $U_q = [0, 2^q - 1]$, and everything that we need for its proof was already described in the proof given above. We have defined the set $B = [0, 2^{q-1} - 2]$ in the proof of the $U_q = [-2^{q-1}+1, 2^{q-1}-1]$ case. If we define B by $B = [0, 2^q-2]$ and follow the proof only from this point with appropriate changes then we get a proof of the $U_q = [0, 2^q - 1]$ case.

2. Most of the difficulties in the proof of Lemma 13 was created by the possibility of an overflow. Therefore if we may assume that the contents of the memory cells are always in a smaller interval, e.g., in $(-2^{\lfloor q/2 \rfloor}, 2^{\lfloor q/2 \rfloor})$, then the proof and the circuit computating $f(x, y)$ become significantly simpler.

Definition. 1. Suppose that, $g_0^{(q)}, ..., g_{\sigma-1}^{(q)}$ are the basic arithmetic functions defined on U_q. Assume further that the arity of $g_i^{(q)}$ is a_i for $i = 0, 1, ..., \sigma - 1$, $q = 1, 2, ...$ Let $\mathcal{L}$ be a firstorder language with equality whose function symbols are $f_0, ..., f_{\sigma-1}$ with arities $a_0, ..., a_{\sigma-1}$, and whose only relation symbol (apart from equality) is the binary relation symbol $\leq$.

2. μ_q will be the unique interpretation of $\mathcal{L}$ so that $\texttt{universe}(\mu_q) = U_q$, $\mu_q(f_i) = g_i^{(q)}$ for $i = 0, 1, ..., \sigma - 1$, and $\mu_q(\leq)$ is the natural ordering of the integers restricted to U_q.

Lemma 14 *Assume that k is a positive integer, $P_1(x_1, ...x_k), ..., P_m(x_1, ..., x_k)$ are propositional formulas of the language $\mathcal{L}$, and $t_1(x_1, ..., x_k), ..., t_{m+1}(x_1, ..., x_m)$ are terms of the language $\mathcal{L}$, containing no other free variables than $x_1, ..., x_k$. For each $q = 1, 2...$, we define a k-ary function $F^{(q)}$ on U_q with values in U_q in the following way. For each $a_1, ..., a_k$ if i is the smallest element of $\{1, ..., k\}$ with $\mu_q \models P_i(a_1, ..., a_k)$ then $F_q(a_1, ..., a_k) = t_i(a_1, ..., a_k)$. If such an integer i does not exist, then $F_q(a_1, ..., a_k) = t_{m+1}(a_1, ..., a_k)$.*

Then the family of functions F_q, $q = 1, 2, ...$ is arithmetic.

Proof. The only atomic formulas that occur in the propositional formulas P_i are of the form $s = t$ or $s \leq t$ where s, t are terms of $\mathcal{L}$. Therefore Lemma 13 and Proposition 9 together imply that for each $i = 1, ..., m$ there exists an arithmetic family of functions $h_i^{(q)}(x_1, ..., x_m)$ so that for all $q = 1, 2, ...$, and for all $a_1, ..., a_m$, we have that $h_i^{(q)}(a_1, ..., a_m) \in \{0, 1\}$ and $h_i^{(q)} = 1$ iff $\mu_q \models P_i(a_1, ..., a_k)$. We define a function $h_{m+1}^{(q)}$ by $h_{m+1}^{(q)}(x_1, ..., x_k) = 1$ for all $x_1, ..., x_k \in U_q$.

Proposition 8 implies that for each $i = 1, ..., m+1$ there is an arithmetic family of functions $\tau_i^{(q)}$ so that for each $q = 1, 2, ...$ and for each $a_1, ..., a_k \in U_q$ we have $\mu_q \models t_i(a_1, ..., a_k) = \tau_i^{(q)}(a_1, ..., a_q)$. The definition of F_q implies that

$$F_q(a_1, ..., a_k) = \sum_{j=1}^{m+1} h_j^{(q)}(a_1, ..., a_k)\tau_j^{(q)}(a_1, ..., a_k) \prod_{i=1}^{j-1}(1 - h_i^{(q)}(a_1, ..., a_k))$$

Therefore, using Proposition 8 and the facts that multiplication and addition are basic arithmetic functions, we get that the family F_q, $q = 1, 2, ...$ is arithmetic. *Q.E.D.*(Lemma 14)

Proof of Lemma 12. The statement of the lemma is a consequence of Lemma 14. Indeed, all of the definitions of the instructions of $\mathcal{M}$ (provided that they do not involve memory cells other than $\texttt{cell}(0), ..., \texttt{cell}(k-1)$), can be expressed by propositional statements using the only atomic formulas of the type $s = t$ and $s \leq t$, where s, t are terms of $\mathcal{L}$. (Here we use that each of the integers $0, 1, ..., k-1$ can be considered 0-ary arithmetic family of functions, for

$q = 1, 2, ...$ E.g., for each $i \in \{0, 1, ..., k-1\}$, $u_q = i$, $q = 0, 1, ...$ is such a family. We get it with composition from the basic arithmetic functions.) The mentioned propositional definitions can be easily transformed in the form required by Lemma 14. *Q.E.D.*(Lemma 12)

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] M. Ajtai, J. Komlós, E. Szemerédi, *An $O(n \log n)$ sorting network.*, 15th STOC. pp. 1-9, 1983.

[3] N. Alon, J. Spencer *The Probabilistic method*, John Wiley & Sons Inc. New York, 1992. 1993.

[4] K. Batcher *Sorting networks and their applications*, AFIPS Spring Joint Computer Conference, Vol. 32, AFIPS Press, Richmond, Va., pp. 307-314, 1968.

[5] I. Damgård, S. Meldgaard, and J. Nielsen, *Perfectly Secure Oblivious RAM Without Random Oracles*, Cryptology ePrint Archive, Report 2010/108, 2010.

[6] O. Goldreich, *Towards a Theory of Software Protection and Simulation by Oblivious RAMs* Proc. of the 19th STOC, pp. 182-194, 1987

[7] O. Goldreich, R. Ostrovsky, *Software Protection and Simulation on Oblivious RAMs* Journal of the Association for Computing Machinery, Vol 43, No 3, May 1996, pp. 431-473.
See also http://www.wisdom.weizmann.ac.il/ oded/PS/soft.ps

[8] R. Ostrovsky, *Efficient Computation on Oblivious RAMs*, Proceedings of the 22nd STOC, pp. 514-523, 1990.

[9] R. Ostrovsky, *Software Protection and Simulation on Oblivious RAMs*, MIT Ph.D. Thesis, Computer Science, May 1992,
see http://www.cs.ucla.edu/ rafail/PUBLIC/09.pdf

[10] D. A. Osvik, A. Shamir, and E. Tromer, *Cache Attacks and Countermeasures: the Case of AES*, Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006,
see also http://people.csail.mit.edu/tromer/papers/cache.pdf

[11] N. Pippenger, M. J.Fischer, *Relations among Complexity Measures*, Journal of the Association for Computing Machinery, Vol 26, No 2, April 1979, pp 361-381.

[12] C. Rackoff and D. Simon, *Cryptographic defense against traffic analysis*, Proceedings of the 25th STOC, pp., 672 - 681, 1993

ECCC ISSN 1433-8092
http://eccc.hpi-web.de