Pebbling and Proofs of Work

Cynthia Dwork[1], Moni Naor[2⋆], and Hoeteck Wee[3⋆⋆]

[1] Microsoft Research, Silicon Valley Campus, dwork@microsoft.com
[2] Weizmann Institute of Science, moni.naor@weizmann.ac.il
[3] University of California, Berkeley, hoeteck@cs.berkeley.edu

Abstract. We investigate methods for providing easy-to-check proofs of computational effort. Originally intended for discouraging spam, the concept has wide applicability as a method for controlling denial of service attacks. Dwork, Goldberg, and Naor proposed a specific memory-bound function for this purpose and proved an asymptotically tight amortized lower bound on the number of memory accesses any polynomial time bounded adversary must make. Their function requires a large random table which, crucially, cannot be compressed.

We answer an open question of Dwork et al. by designing a compact representation for the table. The paradox, compressing an incompressible table, is resolved by embedding a time/space tradeoff into the process for constructing the table from its representation.

1 Introduction

In 1992 Dwork and Naor proposed that e-mail messages be accompanied by easy-to-check *proofs of computational effort* in order to discourage junk e-mail, now known as spam [12], and suggested specific CPU-bound functions for this purpose[4]. Noting that memory access speeds vary across machines much less than do CPU speeds, Abadi, Burrows, Manasse, and Wobber [1] initiated a fascinating new direction: replacing CPU-intensive functions with *memory-bound* functions, an approach that treats senders more equitably.

Memory-bound functions were further explored by by Dwork, Goldberg, and Naor [11], who designed a class of functions based on pointer chasing in a large shared random table, denoted T. We may think of T as part of the definition of their functions. Using hash functions modelled as truly random functions (i.e. 'random oracles'), they proved lower bounds on the amortized number of memory accesses that an adversary must expend per proof of effort, and gave a concrete implementation in which the size of the proposed table is $16MB$.

There are two drawbacks to the use of a large random table in the definition of the function. If the proof-of-effort software is distributed bundled with other software, then the table occupies a large footprint. In addition, for

⋆ Incumbent of the Judith Kleeman Professorial Chair. Research supported in part by a grant from the Israel Science Foundation.
⋆⋆ Work performed at Microsoft Research, Silicon Valley Campus.
[4] A similar proposal was later made by Back in 1997 [6].

users downloading the function (or a function update, possibly necessitated by a substantial growth in cache sizes), downloading such a large table might require considerable connect time, especially if the download is done on a common telephone line. Connection fees, i/o-boundedness, and the possibility of transmission errors suggest that five minutes of local computation, to be done once and for all (at least, until the next update), is preferable to five minutes of connect time. Thus a compact representation of T allows for easy distribution and frequent updates.

These considerations lead to the question of whether there might be a succinct representation of T. In other words, is it possible to distribute a *short program* for constructing T while still maintaining the lower bound on the amortized number of memory accesses? The danger is that the adversary (spammer) might be able to use the succinct description of T to generate elements in cache and on the fly, whenever they are needed, only rarely going to memory.

Roughly speaking, our approach is to generate T using a memory-bound *process*. Sources for such processes are time/space tradeoffs, such as those offered by *graph pebbling*, defined below, and *sorting*. We will use both of these: we exploit known dramatic time/space tradeoffs for pebbling in constructing a theoretical solution, with provable complexity bounds; the solution uses a hash function, modeled by a random oracle in the proof. We also describe a heuristic based on sorting. A very nice property of an algorithm whose most complex part is sorting is that it is easy to program, reducing a common source of implementation errors.

We will focus most of our discussion on the pebbling results. The heuristic based on sorting is described in Section 6. Although our work does not rely on computational assumptions, we nonetheless assume the adversary is restricted to polynomial time, or in any event that a spamming approach that requires superpolynomially many cpu cycles is not lucrative. This raises an interesting observation:

Remark 1. If the adversary were not restricted to polynomial time, then the proof of effort would have to be long. Otherwise, by Savitch's Theorem (relativized to a random oracle), the adversary could find the proof using space at most the square of the length of the proof (since guessing the proof has non-deterministic space complexity bounded by the proof length), which may be considerably smaller than the cache size.

Pebbling can be described as a game played on a directed acyclic graph $D(V, E)$ be with a set $\mathcal{S} \subset V$ of inputs (nodes of indegree 0) and a set $\mathcal{T} \subset V$ of outputs (nodes of outdegree 0). (Eventually we will identify the outputs of D with the elements of the table T.) The player has s pebbles. A pebble may be placed on an input at any time. A pebble may be placed on a node $v \in V \setminus \mathcal{S}$ if and only if every vertex u such that $(u, v) \in E$ currently has a pebble. That is, a non-input may be pebbled if and only if all its immediate predecessors hold pebbles. Finally, a pebble may be removed from the graph at any time. Typically, the goal of the player is to pebble outputs using few moves and using few simultaneous pebbles; that is, efficiently and in such a way that at any time there are few pebbles on the graph.

Pebbling has been the subject of deep and extensive research, and it is in the context of proving lower bounds for computation on random access machines *superconcentrators* were invented (see [18]). These are graphs with large flow: for every set A of inputs and every set B of outputs of the same size, there are vertex-disjoint paths connecting A to B. Valiant [18] showed that every circuit for computing a certain type of transform contains a superconcentrator. Although these did not directly yield lower bounds, they eventually yielded time-space tradeoffs, via pebbling arguments.

Pebbling intends to capture time and space requirements for carrying out a particular computation, defined by the graph – we can think of a non-input as being associated with a function symbol, such as "+" or " ×". For example, a sum can be computed if its summands – as represented by the node's immediate predecessors – have been computed. We can think of placing a pebble on a node as tantamount to storing a (possibly newly computed) value in a register. Time/space tradeoffs for specific computation graphs are obtained by showing that no (time efficient) pebbling strategy exists that uses few simultaneous pebbles. Time/space tradeoffs for problems are obtained by showing that every computation graph for the problem yields a tradeoff.

As noted, in our case, the outputs of the graph will correspond to elements of T. If an output cannot be pebbled (in reasonable time) using few pebbles, i.e., little space, we would like to conclude that considerable time or memory accesses were devoted to finding the value associated with the corresponding output of the graph computation. But perhaps there is a *different* computation that yields the same outputs – a computation unrelated to the computation determined by the dag. In this case obtaining a function output does not imply that significant resources have been expended.

We force the adversary to adhere to the computation schema described by the graph by associating a random oracle with each node. That is, we *label* each non-input with the value obtained by applying the hash function to the labels of the predecessor vertices. (The inputs are numbered 1 through N, and the label of input i is the hash of i.) Using this we show, rigorously, how to convert the adversary's behavior to a pebbling.

Although this intuition is sound, our situation is more challenging: while we associate placing a pebble with storing a value in cache, our adversary is not limited to cache memory, but may use main memory as well; moreover, main memory is very large. How can we adapt the classical time/space tradeoffs to this setting?

We address this by constructing a dag D that, in addition to being hard to pebble, remains hard to pebble even when many nodes and their incident edges are removed. Roughly speaking, given the cache contents, we "knock out" those nodes whose labels can be (mostly) determined by the cache contents, together with their incident edges. We need that the "surviving" graph is still hard to pebble. We will also introduce the concept of *spontaneously generated* pebbles to capture memory reads. We show that pebbling remains hard unless the number of spontaneously generated pebbles is large.

We must ensure that no cut in the graph is knocked out. Intuitively, knocking out a cut in the graph corresponds to reducing the depth of the graph, and therefore reducing the difficulty of pebbling the graph. This leads us to a graph with slightly special structure: it is the concatenation of two pieces. The first is a stack of wide (no small cut) superconcentrators (by stack we mean a sequence of DAGs where the outputs of one are the inputs of the next DAG in the sequence). The description of the second is quite technical, and we defer it until the requirements have been better motivated (Section 4).

2 Complete Description of Our Abstract Algorithm

We now describe our algorithm, *Compact_MBound*, postponing the construction of the graph D to Section 4. The algorithm adds a table generation phase to Algorithm MBound of [11]. Thus the two algorithms are identical except that in the new algorithm the table T is generated by the procedure outlined below, while in the original algorithm the table T is completely random. For the concrete implementation, we use a heuristic for generating the table T, described in Section 6. We then combine this with the concrete implementation of Algorithm MBound proposed in [11].

Algorithm Compact_Mbound uses a collection of hash functions $\mathcal{H}' = \{H_0, H_1, H_2, H_3, H_4\}$. The function H_4 has been described in the Introduction; its role is to force the adversary to adhere to a computation defined by the graph D, and its output is w bits long, where w is the word size. The remaining hash functions are those used in the original Algorithm MBound. In our analysis, we will treat each hash function as a random oracle.

2.1 Building Table T

Both the (legitimate) sender and the receiver must build the table T, but this is done only once and the table T is then stored in main memory. After the table has been built, proofs of effort are constructed and checked using the algorithms of Dwork, Goldberg, and Naor [11], reproduced in Section 2.2 for completeness. We will provide an explicit construction of the graph D used in building T. This means that we may incorporate the algorithm for computing D (and thus T) into the proof-of-effort software.

The algorithm for constructing T first computes the graph D which has N inputs, N outputs and constant indegree d, and then numbers the input vertices $1, 2, \ldots, N$ and the output vertices $N+1, N+2, \ldots, 2N$. Next, each vertex of V is labeled with a w-bit string in an inductive fashion, beginning with the input vertices:

1. Input i is labeled $H_4(i)$, for $1 \leq i \leq N$.
2. For vertex $j \notin [N]$, let vertices $i_1 < \cdots < i_d$ be the predecessors of vertex j. Then vertex j is labeled $H_4(\mathit{label}(i_1), \ldots, \mathit{label}(i_d), j)$.

The entries of T correspond to the labels of the output vertices, namely:

$$T[i] = label(N + i),\ i = 1, 2, \ldots, N.$$

Once the table T has been computed and stored, the graph D and the node labels may be discarded.

2.2 Computing and Checking Proofs of Effort

The algorithm described here is due to [11]. It uses a modifiable array A, initialized for each trial. The adversary's model, described in Section 2.3, restricts the size of A: if w is the size of a memory word and b is the number of bits in a memory block (or cache line), then the algorithm requires that $|A|w > b$ bits. A word on notation: For arrays A and T, we denote by $|A|$ (respectively, $|T|$) the number of elements in the array. Since each element is a word of w bits, the numbers of *bits* in these arrays are $|A|w$ and $|T|w$, respectively.

At a high level, the algorithm is designed to force the sender of a message to take a random walk "through T," that is, to make a series of random-looking accesses to T, each subsequent location determined, in part, by the contents of the current location. Such a walk is called a *path*. Typically, the sender will have to explore many different paths until a path with certain desired characteristics is found. Such a path is called *successful*, and each path exploration is called a *trial*. Once a successful path has been identified, information enabling the receiver to check that a successful path has been found is sent along with the message.

The algorithm for computing a path in a generic trial is specified by two parameters ℓ (path length) and e (effort), and takes as input a message m, sender's name (or address) S, receiver's name (or address) R, and date d, together with a trial number k:

Initialization:
  $A = H_0(m, R, S, d, k)$
Main Loop: Walk for ℓ steps (ℓ is the path length):
  $c \leftarrow H_1(A)$
  $A \leftarrow H_2(A, T[c])$
Success occurs if after ℓ steps the last e bits of $H_3(A)$ are all zero.

A legitimate proof of effort is a 5-tuple (m, R, S, d, k) along with the value $H_3(A)$ for which success occurs. This may be verified with $O(\ell)$ work by just exploring the path specified by k and checking that the reported hash value $H_3(A)$ is correct and ends with e zeroes. The value of $H_3(A)$ is added to prevent the adversary from simply guessing k, which has probability $1/2^e$ of success.

An honest sender computes a proof of effort by repeating path exploration for $k = 1, 2, \ldots$ until success occurs. The probability of success for each trial is $1/2^e$, so the expected amount of work for the honest sender is $O(2^e\ell)$. The main technical component in [11] is showing that $\Omega(2^e\ell)$ work is also *necessary* (for a random T).

2.3 The adversary's model

We assume an adversary's computational model as in [11]. The adversary is assumed to be limited to a "standard architecture" as specified below:

1. There is a large memory, partitioned into m blocks (also called cache lines) of b bits each;
2. The adversary's cache is small compared to the memory. The cache contains at most s (for "space") *words*; a cache line typically contains a small number (for example, 16) of words;
3. Although the memory size is large compared to the cache, we assume that m is still only polynomial in the largest feasible cache size s;
4. Each word contains w bits (commonly, $w = 32$);
5. To access a location in the memory, if a copy is not already in the cache (a *cache miss* occurs), the contents of the block containing that location must be brought into the cache – a *fetch*; every cache miss results in a fetch;
6. We charge one unit for each fetch of a memory block. Thus, if two adjacent blocks are brought into cache, we charge two units (there is no discount for proximity at the block level);
7. Computation on data *in the cache* is essentially *free*. By not (significantly) charging the adversary for this computation, we are increasing the power of the adversary; this strengthens the lower bound.

3 Pebbling

The goal in pebbling is to find a strategy to pebble all the outputs while using only a few pebbles simultaneously and not too many steps (pebble placements). Pebbling has received much attention, in particular in the late seventies and early eighties, as a model for space bounded computation (as well as other applications, such as the relative power of programming languages) [9, 13, 14, 16].

A directed acyclic graph with bounded indegree, N inputs, and N outputs is an N-*superconcentrator* if for any $1 \leq k \leq N$ and any sets $\mathcal{S}'$ of inputs and $\mathcal{T}'$ of outputs, both of size k, there are k vertex-disjoint paths connecting $\mathcal{S}'$ to $\mathcal{T}'$. Thus, superconcentrators are graphs with excellent flow. (Note that we do not assume the need to specify which input is connected to which output.)

The following classical results are relevant to our work. Here m denotes the number of vertices in the graph.

– Stacks of superconcentrators yield graphs with a very sharp tradeoffs: To pebble all the outputs of these graphs with fewer than N pebbles requires time exponential in the depth, *independent of the initial configuration of the pebbles* [13].
– Constant-degree constructions of linear-sized superconcentrators of small depth were given in [18, 15, 10]. The construction of minimum known density is by Alon and Capalbo [4].

The Basic Lower Bound Argument. Many proofs of pebbling results rely on the following so-called *Basic Lower Bound Argument* [16, 13]. The claim of the Basic Lower Bound Argument is that to pebble $s + 1$ outputs of a superconcentrator with *any* initial placement of at most s pebbles requires the pebbling of $N - s$ inputs, independent of the initial configuration of the s pebbles.

To see this, suppose that fewer than $N - s$ inputs are pebbled. Then there exists a set S'' of $s + 1$ inputs that do not receive pebbles. By the superconcentrator property these $s + 1$ inputs are connected via vertex-disjoint paths to the target set of $s+1$ outputs that should be pebbled. Every one of these paths must at some point receive a pebble, else not all the target outputs can be pebbled. Since a node cannot be pebbled without pebbling all its ancestors, it follows that every input in our set of size $s + 1$ must receive a pebble at some point, contradicting the assumption.

3.1 Converting the Adversary's Moves to a Pebbling

The adversary does not define its operation in terms of pebbling but instead we assume that we (the provers of the lower bound) can follow its memory accesses and the applications of the functions of $\mathcal{H}'$ and in particular H_4. We now describe how the adversary's actions yield a pebbling of the graph. The pebbling is determined by an *off-line* inspection of the adversary's moves, i.e., following an execution of the adversary it is possible to describe the pebbling that occurred. Hence we call it *ex post facto* pebbling:

Placing initial pebbles If H_4 is applied with $label(j')$ as an argument, and $label(j')$ was not computed via H_4, then we consider j' to have a pebble in an initial configuration. We sometimes refer to these as *spontaneously generated* pebbles.

Placing a pebble If H_4 is applied to i for some $1 \leq i \leq N$, then place a pebble on node i (recall the inputs are vertices $1, \ldots, N$, so node i in this case is an input vertex). Let j be a non-input vertex (so $j > N$), and let $i_1 < \cdots < i_d$ be the predecessors of vertex j. If H_4 is applied to $(label(i_1), \ldots, label(i_d), j)$, where $label(i_b)$ is the correct label of vertex i_b, $1 \leq b \leq d$, then place a pebble on vertex j.

Removing a pebble A pebble is removed as soon as it is not needed anymore. Here we use our clairvoyant capabilities and remove the pebble on node j' right after a call for H_4 with the correct value of $label(j')$ as one of the arguments if (i) $label(j')$ is not used anymore or (ii) $label(j')$ is computed again before it is used as an argument to H_4. That is, before the next time $label(j')$ appears as an argument to H_4 it also appears as the result of computing H_4 (the output of H_4).

We may relate the ex post facto pebbling strategy to the adversary's strategy as follows:

– Placing a pebble corresponds to making an oracle call to H_4. Hence, a lower bound on the number of (placement) moves in the pebbling game yields a lower bound on the number of oracle calls and thus the amount of work done by the adversary.

– The initial (spontaneously generated) pebbles correspond to the values of H_4 that the adversary "learns" without invoking H_4. Intuitively, this information must come from the cache contents and memory fetches. Therefore, we would expect that if the adversary has a cache of s words and fetches z bits from memory, then the adversary is limited to at most $s + z/w$ initial pebbles, since each pebble corresponds to a w-bit string.

The following lemma formalizes our intuition relating the number of pebbles used in the ex post facto pebbling to the cache size of the adversary and the number of bits the adversary fetches from memory. It says that with very high probability the ex post fact pebbling uses only $s + z/w$ simultaneous pebbles. The intuition is that if more pebbles are used, then somehow the sw bits in the cache and the additional z bits obtained from memory are being used to reconstruct $s + z/w + 1$ labels, or $sw + z + w$ bits.

Lemma 1. *Consider an adversary that operates for a certain number of steps and where:*

– *the adversary is using a standard architecture as specified in Section 2.3 with a cache of s words of size w; and*
– *the adversary brings from memory at most z bits.*

Then with probability at least $1 - 2^{-w}$ the maximum number of pebbles at any given point in the ex post facto pebbling is bounded by $s + z/w$. The probability is over $\mathcal{H}'$.

Proof. We need the following simple observation:

Claim 1. *Let $b_1 \dots b_u$ be independent unbiased random bits and let $k \leq u$. Suppose we have a (randomized) reconstruction procedure that, given a hint of length $B < k$ (which may be based on the value of $b_1 \dots b_u$), produces a subset $S \subset \{1, \dots u\}$ of k indices and a guess of the values of $\{b_i \,|\, i \in S\}$. Then the probability that all k guesses are correct is at most $2^B/2^k$, where the probability is over the random variables and the coin flips of the hint generation and the reconstruction procedures.*

Proof (of claim). Fix an arbitrary sequence of random choices for the reconstruction procedure. Each fixed hint yields a choice of S and a guess of the bits of S. For any fixed hint, the probability, over choice of $b_1, \dots, b_u$, that all the guessed values are consistent with the values of the elements of S is 2^{-k}. Therefore, by summing over all hints, the probability that there exists a hint yielding a guess consistent with the actual value of $b_1, \dots, b_u$ is 2^{B-k}. □

Suppose there are $s + z/w + 1$ pebbles at some point in the ex post facto pebbling. We apply the claim to bound the probability that this occurs, as follows: the independent unbiased random bits $b_1, \dots, b_u$ are from the truth table of $\mathcal{H}'$; the hint consists of three parts:

1. The cache contents $C \in \{0,1\}^{sw}$
2. The bits brought from main memory (at most z)
3. The values of the functions in $\mathcal{H}'$ needed to simulate the adversary *except those values of H_4 corresponding to the initial pebbles.*

The output of the reconstruction procedure is all the values of the functions in $\mathcal{H}'$ given in the hint, plus the labels of the $s + z/w + 1$ pebbled nodes. The reconstruction procedure works by simulating the adversary (up to the point where there are $s + z/w + 1$ pebbles) and outputting the labels of the pebbled nodes as the values are generated. The difference between the length of the output (in bits) and the length of the hint (in bits) is w, so the probability is bounded above by 2^{-w}.

□

4 Description of Our Graphs

Call our graph D and let it be composed from two pieces D_1 and D_2. We are interested in a graph with a small number of nodes and edges, since each node corresponds to an invocation of H_4 and the number of edges corresponds to the total size of inputs in the H_4 calls. We are less concerned with the depth of the graph.

The dag D has $N = |T|$ inputs and outputs. It is constructed from two dags D_1 and D_2 via concatenation; that is:

- Inputs of D_1 are the inputs of D.
- The outputs of D_1 are the inputs of D_2.
- Outputs of D_2 are the outputs of D.

The properties we require from the two dags are different. In particular we allow the spammer more pebbles for the D_1 part. For each of D_1 and D_2 we first describe the properties needed (in terms of pebbling) and then mention constructions of graphs that satisfy these requirements.

The Dag $D_1 = (V_1, E_1)$: We require an almost standard pebbling lower bound property: for a certain β to be determined later, pebbling any $m > s + 2\beta b/w$ outputs of D_1 is either impossible or at the very least requires exponential (in the depth of D_1) time (making it infeasible) provided we are constrained to:

1. start from any initial setting of at most $s' = s + 2\beta b/w$ pebbles. The number of pebbles in the initial configuration comes from two sources: the cache size in words (s) and the bits brought from memory during a given interval, divided by the word size w. In Section 5 the number of blocks brought from memory will be denoted 2β, containing a total of $2\beta b$ bits. (In the language of Lemma 1, $z = 2\beta b$.) Hence $s' = s + 2\beta b/w$.
2. use at most s pebbles during the pebbling itself; that is, of the initially placed pebbles, some are designated as *permanent*, and do not move. Only s pebbles may be moved, and these s may be moved repeatedly.

Constructing D_1: One way to obtain such a graph is to consider a stack of $\ell_1 \in \omega(\log |T|)$ N-superconcentrators (that is, N inputs and N outputs). Then following the work of [13] (Section 4) we know that D_1 has the desired property: independent of the initial configuration, the time to pebble $m > s'$ outputs requires time at least exponential in the depth, hence, superpolynomial in $|T|$. This means that when we consider in Section 5 an interval of computation by the spammer, then by appealing to Lemma 1 we can argue that either (i) fewer than m outputs of D_1 were pebbled or (ii) at least $mw - 2\beta b$ bits were brought from main memory during this interval (which suffices to show high amortized memory accesses). This follows from the randomness of the labels of V_1 (that is, the randomness of H_4), as discussed above.

Since there are linear-sized superconcentrators [18, 15, 4], this means the size of V_1 can be some function in $\omega(|T| \log |T|)$. Also these constructions are explicit, so we have an *explicit* construction of D_1.

Remark 2. An alternative graph to the stack of superconcentrators is that of Paul, Tarjan and Celoni [14], where the number of nodes is $O(N \log N)$.

The Dag $D_2 = (V_2, E_2)$: The property needed for D_2 is that even if a significant fraction of the vertices fail it should be very hard to disconnect small sets of surviving outputs from the surviving inputs. (For now, think of a failed node as one whose label is largely determined by the cache contents.)

The vertices V_2 are partitioned into layers $L_1, L_2, \ldots L_{\ell_2}$ of size N. Suppose that nodes in V_2 fail but we are guaranteed that from each layer L_i at least a δ fraction of nodes survives. The condition on the surviving graph can be expressed as follows: There exists a set S' of the inputs and a set T' of the outputs both of size $\Omega(N)$, such that, for any set $U \subset T'$ of x outputs, where the bound x is derived from the proof of Algorithm Mbound, to completely disconnect U from all of S' by removing nodes requires either cutting $\Omega(x)$ vertices in *some* level not including the input, or cutting $\Omega(N)$ inputs.

Constructing D_2: One way to construct D_2 is using a stack of bipartite expanders on N nodes, where we identify the left set of one expander with the right set of the other, except for the inputs and outputs of D_2, which are identified with the leftmost and rightmost sets respectively; the orientation of the edges is from the inputs to the outputs. Expanders are useful for us for two reasons: (i) they do not have small cuts and (ii) they have natural fault tolerance properties. In particular, Alon and Chung [5] have shown that in any good enough expander if up to some constant (related to the expansion) fraction of nodes are deleted, then one can still find a smaller expander (of linear size) in the surviving graph (see also Upfal [17]).

Consider now the dag obtaining by stacking $\ell_2 = O(\log |T|)$ bipartite expanders where each side has $N = |T|$ nodes. We give here an intuitive explanation of why disconnecting a relatively small set U of surviving outputs from the surviving inputs requires deleting $|U|/2$ vertices at some level. Following the argument in [5], there exists a subgraph F of the surviving graph with the following property: every layer of F contains $\delta N/2$ vertices, and the bipartite graph induced by any two consecutive layers in F satisfies a vertex expansion property (in the direction from the output nodes towards the input nodes) with expansion factor 2 for subsets of size at most $\delta N/16$, say. Consider any set U of size $o(N)$ outputs in F. Clearly, by deleting U we can disconnect it from the inputs. Suppose we delete at most $|U|/2$ output vertices in F. Then, there are at least $|U|/2$ vertices left amongst the output nodes, which are connected to at least $|U|$ vertices in level $\ell_2 - 1$ in F. Again, if we delete at most $|U|/2$ output vertices in level $\ell_2 - 1$, then U must be connected to at least $|U|$ vertices in level $\ell_2 - 2$. Continuing this argument, we may deduce that U must be connected to at least $|U|/2$ input nodes in F, unless we delete at least $|U|/2$ nodes at some level. To ensure that disconnecting $|U|$ inputs is insufficient we use an additional property of D_2: the surviving subgraph D_2' contains a substantial superconcentrator. Restricting out attention to output sets U of the surviving superconcentrator suffices for our lower bound proof.

We conclude that the total number of nodes in V_2 can be $O(|T|\log|T|)$ and thus the dominating part is D_1. Also we have explicit construction of expanders and hence of D_2.

5 An Amortized Lower Bound on Cache Misses

In this section we prove that any spammer limited to a standard architecture (as specified in Section 2.3) and trying to generate many different proofs of computational efforts according to Algorithm Compact_Mbound presented in Section 2 (i.e. the verifier follows the algorithm there, while the spammer is free to apply any algorithm), will, with high probability over choices of the random oracles and the choices made by the adversary, have a large amortized number of cache misses.

5.1 The Lower Bound

We are now ready to state the main theorem:

Theorem 1. *Fix an adversary spammer $\mathcal{A}$. Consider an arbitrarily long but finite execution of $\mathcal{A}$'s program – we don't know what the program is, only that $\mathcal{A}$ is constrained to use an architecture as described in Section 2 and that its computation of H_1 and H_2 has to be via oracle calls.*

Under the following additional conditions, the expectation, over choice of the hash functions $\mathcal{H}'$, and the coin flips of $\mathcal{A}$, of the amortized complexity of generating a proof of effort that will be accepted by a verifier is $\Omega(2^e\ell)$. Note that in [11], "choice of T" meant choice of the random table T. In our case, "choice of T" means choice of the hash function H_4.

- $|T| \geq 2s$ *(recall that the cache contains s words of w bits each)*
- $|A|w \geq bs^{1/5}$ *(recall that b is the block size, in bits).*
- $\ell > 8|A|$

- *The total amount of work by the spammer (measured in oracle calls) per successful path is no more than $2^{o(w)}2^e\ell$ and no more than 2^{ℓ_1}, where ℓ_1 is the depth of the dag D_1.*
- *ℓ is large enough so that the spammer cannot call the oracle 2^ℓ times.*

The amortized cost of a proof of effort is the sum of the costs of the individual proofs divided by the number of proofs. The theorem says that

$$E_{\mathcal{H}',\mathcal{A}}[\text{amortized cost of proof of effort}] = \Omega(2^e\ell) \tag{1}$$

Remark 3. As noted in [11], if (for some reason) it must be the case that $|A| \leq O(b/w)$, then the lower bound obtained is $\Omega(2^e\ell/\log s)$. Also, as noted in [11], the theorem holds if *expected* amortized cost (over $\mathcal{H}'$ and flips) is replaced with "with high probability."

Our proof follows the structure of the proof in [11]; naturally, however, we must make several modifications since T is no longer random. We will describe the key lemma in the original proof, and sketch the proof given in [11]. We will then state and sketch the proof of the new version of the key lemma, yielding a proof of Theorem 1. We start with a simple lemma from [11]:

Lemma 2 ([11]). *The expected amortized number of calls to H_1 and H_2 per proof of effort that will be accepted by a verifier is $\Omega(2^e\ell)$. The expectation is taken over T, $\mathcal{A}$, and $\mathcal{H} = \mathcal{H}' \setminus \{H_4\}$.*

In our case an analogous lemma holds (with exactly the same proof). This time, the expectation is taken over $\mathcal{A}$ and $\mathcal{H}'$ (recall that in the current work T is defined by H_4 and the dag D).

The execution is broken into intervals in which, it will be shown, the adversary is forced to learn a large number of elements of T. That is, there will be a large number of scattered elements of T which the adversary will need in order to make progress during the interval, and very little information about these elements is in the cache at the start of the interval. The proof holds even if the adversary is allowed during each interval, to remember "for free" the contents of all memory locations fetched during the interval, provided that at the start of the subsequent interval the cache contents are reduced to sw bits once again.

For technical reasons the proof focuses on the values of A only in the second half of a path. Recall that A is modified at each step of the Main Loop; intuitively, since these modifications require many elements of T, these "mature" A's cannot be compressed. The definition of an interval allows focusing on progress on paths with "mature" A's.

A *progress call* to H_1 is a call in which the arguments have not previously been given to H_1 in the current execution. Let $n = s/|A|$. A progress call is *mature* if it is the jth progress call of the path, for $j > \ell/2$ (recall that ℓ is the length of a path).

Let k be a constant determined in [11]. An *interval* is defined by fixing an arbitrary starting point in an execution of the adversary's algorithm (which may involve the simultaneous exploration of many paths), and running the execution until kn mature progress calls (spread over any number of paths) have been made to oracle H_1.

At any point in the computation, the *view* of the spammer is T together with the parts of the oracles $\mathcal{H}$ that the spammer has explicitly invoked. Intuitively, the view contains precisely that information which can have affected the memory of the spammer. Since T is (when T is random, and in any case, could be) stored in memory, we consider it part of the view.

We now state the key lemma from [11]; recall that in that setting T is random.

Lemma 3. *There is a constant $k \geq 1$ where the following is true. Fix any integer i, the "interval number". Choose T and $\mathcal{H}$, and coin flips for the spammer. Run the spammer's algorithm, and consider the ith interval. The expected number of memory accesses made during this interval is $\Omega(n)$, where the expectation is taken over the choice of T, the functions $\mathcal{H}$, and the coin flips of the spammer. That is,*

$$E_{T,\mathcal{H},\mathcal{A}}[\text{number of memory accesses}] = \Omega(n) \tag{2}$$

Note that between intervals the adversary is allowed to store whatever it wishes into the cache, taking into account all information it has seen so far, in particular, the table T and the calls it has made to the hash functions.

It is an easy consequence of this lemma that the amortized number of memory accesses to find a successful path is $\Omega(2^e \ell)$. This is true since by Lemma 2, success requires an expected $\Omega(2^e \ell)$ mature progress calls to H_1, and the number of intervals is the total number of mature progress calls to H_1 during the execution, divided by kn, which is $\Omega(2^e \ell / n)$. (Note that we have made no attempt to optimize the constants involved.)

5.2 Sketch of Proof of Key Lemma for Random T

We give here a slightly inaccurate but intuitive sketch of the key steps of the proof in [11] of Lemma 3.

The spammer's problem is that of *asymmetric communication complexity* between memory and the cache. Only the cache has access to the functions H_1 and H_2 (the arguments must be brought into cache in order to carry out the function calls). The goal of the (spammer's) cache is to perform *any kn* mature progress calls. Since by definition the progress calls to H_1 are calls in which the arguments *have not previously been given to H_1 in the current execution*, we can assume the values of H_1's responses on these calls are uniform over $1, \ldots, |T|$ given all the information currently in the system (memory and cache contents and queries made so far). The cache must tell the memory which blocks are needed for the subsequent call to H_2. The cache sends the memory $\beta \log m$ bits to specify the block numbers (which is by assumption $o(n \log m)$ bits), and gets in return βb bits altogether from the memory. The key to the proof is, intuitively, that the relatively few possibilities in requesting blocks by the cache imply that many different elements of T specified by the indices returned by the kn mature calls to H_1 must be derived using the same set of blocks. This is shown to imply that more than s elements of T can be reconstructed from the cache contents alone, which is a contradiction given the randomness of T.

It is first argued that a constant δ fraction of elements of T are largely undetermined by the contents of the cache. This is natural, since T is random and the cache can hold at most half the bits needed to represent T. For simplicity, assume that elements that are largely undetermined are in fact completely undetermined, that is, there is simply no information about these in the cache. Call these completely undetermined elements T'.

Simplifying slightly, it is next argued that, for a constant k to be determined later, if one fixes any starting point in the execution of the spammer's algorithm, and considers all oracle calls from the starting point until the knth call to H_1, there will be at least $5n$ pairs of calls to H_1 and H_2 on the same path; that is, H_2 is called on the index determined by the call to H_1.

Intuitively, this observation implies that, since the calls to H_1 return random indices into T, many of the elements of T selected by these invocations will be in T'. That is, there will be no information about them in the cache, and the spammer will have to go to memory to resolve them.

Let β be the average number of blocks sent by the main memory to the cache during an interval. Assume for the sake of contradiction that for at least half the *kn-tuples* of elements selected by H_1, the spammer makes only $2\beta = o(n)$ memory accesses, even though it needs $\Omega(n)$ elements in T' and about which it has no knowledge.

Unfortunately, this assumption of $o(n)$ memory accesses does not yield a contradiction: as a memory access fetches an entire block, which contains multiple words, the total number of bits retrieved (βb) is not necessarily less than the total number of bits needed (at least nw). Indeed, if $\beta \geq nw/b$ there is no contradiction, and it is *not* assumed that w/b is a fixed constant.

The contradiction is derived by using the fact that some set of 2β blocks suffices to reconstruct *many different* possible kn-tuples. This is immediate from a pigeonhole argument (since there are roughly $|T|^{kn}$ kn-tuples and $m^{\beta} = |T|^{O(\beta)}$ choices of β blocks, since we assume that the memory contains $\mathrm{poly}(|T|)$ words). Let $\mathcal{G}'$ denote the largest such set of kn-tuples.

Let Σ denote the union over tuples in $\mathcal{G}'$, of the set of elements in the tuple. That is, Σ contains every element that appears in $\mathcal{G}'$. It is possible to show that $|\Sigma|$ is large, i.e. when the entries are measured in terms of the missing bits, then the total number is $\omega(\beta b)$.

Note that if $\mathcal{G}'$ is known to the cache party, then intuitively, by sending the 2β blocks the memory transmits all $p \in \Sigma$ to the cache: to reconstruct any given p, the cache chooses any kn-tuple in $\mathcal{G}'$ containing p, acts as if the calls to H_1 returned the indices (in T) of the elements in this tuple, and runs the spamming program, extracting p in the process. However, $\mathcal{G}'$ is not known to the cache, since it may depend on the full memory content.

At this point, [11] argues that there is a small collection of "*mighty*" tuples, with the property that each tuple in the collection enables the transmission of "too many" elements in Σ. That is, there exists a too large set U of elements reconstructible from too few mighty tuples. This yields an information-theoretic argument that too many bits are obtained from too few. In the sequel, we let $x = |U|$. In [11] it is shown that setting $x = 4\beta b/w$ yields the desired contradiction. We do not repeat that proof here, but we use the same proof, and so, the same value of x.

This concludes the high-level sketch of the proof in [11] for the key lemma in the case that T is random.

5.3 Lower Bound when T has a Compact Representation

The new key lemma for the lower bound proof is given below. We then sketch those aspects of the proof germane to the case of the compact representation of T. We let $s = |S|$. This is the size of the cache, in words.

Lemma 4. *There is a constant $k \geq 1$ where the following is true. Fix any integer i, the "interval number". Choose $\mathcal{H}'$ and coin flips for the spammer. Run the spammer's algorithm, and consider the ith interval. The expected number of memory accesses made during this interval is $\Omega(n)$, where the expectation is taken over the choice of $\mathcal{H}'$ and the coin flips of the spammer. That is,*

$$E_{\mathcal{H}',\mathcal{A}}[\text{number of memory accesses}] = \Omega(n) \tag{3}$$

Consider an ex post facto pebbling on D induced by the adversary's execution during an interval, obtained as described in Section 3.1.

Either $\Omega(N)$ of the nodes at the level common to D_1 and D_2 been pebbled during the interval or not. In the first case, i.e., the case in which a constant fraction of the nodes at this level have been pebbled, by the discussion in Section 4, many blocks must be brought from memory. So if we are at an execution of a *good* tuple (one for which the adversary goes to memory at most $2\beta b/w$ times) we can conclude that this did not happen.

In the second case, i.e., the case in which $o(N)$ inputs of D_2 are pebbled, we will use the fault-tolerant flow property of D_2 to show that much information will have to be brought from memory *for some layer of D_2*.

As in the previous section, fix the cache contents C and consider the large set $\mathcal{G}'$ of tuples all utilizing the same set of of blocks from memory. The next claim (proved using Sauer's Lemma) guarantees that in each layer of D_2, there is a constant fraction δ of nodes for each which the label is mostly unknown:

Claim 2. *Fix cache contents $C \in \{0,1\}^{sw}$. For any $1 \leq j \leq \ell_2$, for $\gamma, \delta \geq 0$, consider the event that there exists a subset of the entries of L_j, called L'_j, of size at least $\delta|T|$, such that for each node i in L'_j there is a set S_i of $2^{\gamma w}$ possible values for label(i) and all the S_i's are mutually consistent with the cache contents C. Then there exist constants $\gamma, \delta > 0$ such that the probability of this event, over choice of the hash functions $\mathcal{H}'$, is high.*

Note that we applied the Lemma for each level individually and we cannot assume that the missing labels of different layers are necessarily independent of each other. To derive a contradiction we need to argue that there is some level where it is possible learn too many undetermined values from too few hints.

The fault tolerance property of D_2 tells us that even if we delete the remaining $(1-\delta)$ fraction of nodes (whose values may be determined from the cache contents and without access to H_4) from each layer, the surviving graph contains a large surviving superconcentrator, call it F, with excellent flow: to disconnect any set X of x outputs from all the inputs requires removing $\Omega(x)$ vertices at *some* level other than the input level to F, or disconnecting $\Omega(N)$ nodes from the input level (in the latter case we appeal to the properties of D_1).

The set X of outputs is obtained as in Section 5.2, from the union of the collection of *mighty* tuples in $\mathcal{G}'$. The collection covers a set X of unknown entries in T, and that X contains more bits than the $2\beta b$ bits of information brought from memory, ($|X| = x$ is roughly the size of $|\Sigma|$). Since each element in X is covered by some tuple in the collection, when the spammer $\mathcal{A}$ is initiated with that tuple, the ex post facto pebbling process must place a pebble on all paths from the inputs of F to that node. Therefore the union of the pebbles placed by $\mathcal{A}$ on all tuples in the collection disconnects X from the inputs F. By the properties of D_2 this means that $\Omega(x)$ spontaneously generated pebbles were placed at some level. A careful choice of x, following the argument in [11], yields a contradiction: too many bits from too few.

6 A Heuristic Based on Sorting

We now present an alternative construction of the table T, designed with an eye toward simplicity of definition. Our concrete heuristic is based, intuitively, on the known time/space lower bound tradeoff for sorting of Borodin and Cook [8]. However, as opposed to the pebbling results which we were able to convert into lower bound proofs, here we are left with a scheme with a conjectured lower bound only.

1. T is initialized to $T[i] = H_4(i)$, $1 \le i \le N$.
2. Repeat m times:
   Sort T.
   $T[i] := H_4(i, T[i])$.

Inserting i guarantees that collisions do not continue to be mapped to the same value (otherwise the number of distinct values in T could dwindle in successive applications).

The number of iterations m of the loop should be as large as possible while the loop can still be considered to take a “reasonable” amount of time on a relatively slow machine. Given current technology, to defeat a 16MB cache, we take the number of elements in T to be $n = 2^{23}$, where each element is a single 32-bit word. Thus T requires 2^{25} bytes $= 32$MB.

Note that after applying $H_4(i, T[i])$ we have a pretty good idea where this this value will end up after the next sorting phase, up to $\sqrt{|T|}$ roughly. However without actually sorting there does not seem to be a way to find the exact location, and we conjecture that the uncertainty increases with the number of iterations. We would therefore like the number of iterations of the loop to well exceed $\log_2 |T|$, say 40 for a 32MB table. The hash function H_4 may be instantiated with whichever function is considered 'secure' at the time of deployment. If this is considered too costly, then the "best" function that will allow 40 iterations in the desired running time should be used (in general we prefer more iterations than a more secure function).

It would be very interesting to see whether the time/space lower bounds known for sorting [8] and recent advances in space lower bounds [2, 3, 7] can be applied for the sorting heuristic in order to obtain lower bounds on the spammer's work.

References

1. M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. In *Proc. 10th NDSS*, 2003.
2. M. Ajtai. Determinism versus non-determinism for linear time RAMs. In *Proc. 31st STOC*, pages 632–641, 1999.
3. M. Ajtai. A non-linear time lower bound for boolean branching programs. In *Proc. 40th FOCS*, pages 60–70, 1999.
4. N. Alon and M. Capalbo. Smaller explicit superconcentrators. *Internet Mathematics*, 1(2):151–163, 2003.
5. N. Alon and F. Chung. Explicit constructions of linear-sized tolerant networks. *Discrete Math*, 72:15–20, 1988.
6. A. Back. Hashcash – a denial of servic counter-measure. Available at `http://www.cypherspace.org/hashcash/hashcash.pdf`.
7. P. Beame, M. E. Saks, X. Sun, and E. Vee. Super-linear time-space tradeoff lower bounds for randomized computation. In *Proc. 41st FOCS*, pages 169–179, 2000.
8. A. Borodin and S. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11(2):287–297, 1982.
9. S. Cook. An observation on time-storage trade off. *JCSS*, 9(3):308–316, 1974.
10. D. Dolev, C. Dwork, N. Pippenger, and A. Wigderson. Superconcentrators, generalizers, and generalized connectors with limited depth. In *Proc. 15th STOC*, pages 42–51, 1983.
11. C. Dwork, A. V. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In *Advances in Cryptology – CRYPTO'03*, pages 426–444, 2003.
12. C. Dwork and M. Naor. Pricing via processing, or, combatting junk mail. In *Advances in Cryptology – CRYPTO'92*, pages 139–147, 1993.
13. T. Lengauer and R.E. Tarjan. Asymptotically tight bounds on time-space trade-offs in a pebble game. *JACM*, 29(4):1087–1130, 1982.
14. W. Paul, R. E. Tarjan, , and J. R. Celoni. Space bounds for a game on graphs. In *Proc. 8th STOC*, pages 149–160, 1976.
15. N. Pippenger. Superconcentrators. *SIAM J. Comput.*, 6(2):298–304, 1977.
16. M. Tompa. Time-space tradeoffs for computing functions, using connectivity properties of their circuits. *JCSS*, 20(2):118–132, 1980.
17. E. Upfal. Tolerating linear number of faults in networks of bounded degree. In *Proc. 11th PODC*, pages 83–89, 1992.
18. L. G. Valiant. On non-linear lower bounds in computational complexity. In *Proc. 7th STOC*, pages 45–53, 1975.