# On Memory-Bound Functions for Fighting Spam

Cynthia Dwork* Andrew Goldberg† Moni Naor‡

August 15, 2003

**Abstract**

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. They proposed specific CPU-bound functions for this purpose. Burrows suggested that, since memory access speeds vary across machines much less than do CPU speeds, *memory-bound* functions may behave more equitably than CPU-bound functions; this approach was first explored by Abadi, Burrows, Manasse, and Wobber [8].

We further investigate this intriguing proposal. Specifically, we

1. Provide a formal model of computation and a statement of the problem;
2. Provide an abstract function and prove an asymptotically tight amortized lower bound on the number of memory accesses required to compute an acceptable proof of effort; specifically, we prove that, on average, the sender of a message must perform many unrelated accesses to memory, while the receiver, in order to verify the work, has to perform significantly fewer accesses;
3. Propose a concrete instantiation of our abstract function, inspired by the RC4 stream cipher;
4. Describe techniques to permit the receiver to verify the computation with *no* memory accesses;
5. Give experimental results showing that our concrete memory-bound function is only about four times slower on a 233 MHz settop box than on a 3.06 GHz workstation, and that speedup of the function is limited even if an adversary knows the access sequence and uses optimal off-line cache replacement.

## 1 Introduction

Unsolicited commercial e-mail, or spam, is more than just an annoyance. At two to three *billion* daily spams worldwide, or close to 50% of all e-mail, spam incurs huge infrastructure costs, interferes with worker productivity, devalues the internet, and is ruining e-mail.

*Microsoft Research, SVC, 1065 L'Avenida Mountain View, CA 94043, USA; email: dwork@microsoft.com.

†Microsoft Research, SVC, 1065 L'Avenida Mountain View, CA 94043, USA; email: goldberg@microsoft.com.

‡Incumbent of the Judith Kleeman Professorial Chair, Department of Computer Science and Applied Math, The Weizmann Institute of Science, Rehovot 76100, Israel; email: naor@wisdom.weizmann.ac.il. Research supported in part by a grant from the Israel Science Foundation.

This paper focuses on the computational approach to fighting spam, and, more generally, to combating denial of service attacks, initiated by Dwork and Naor [13] (also discussed by Back; see [20, 11]). The basic idea is:

> "If I don't know you and you want to send me a message, then you must prove that you spent, say, ten seconds of CPU time, just for me and just for this message."

The "proof of effort" is cryptographic in flavor; as explained below, it is a moderately hard to compute (but very easy to check) function of the message, the recipient's address, and a few other parameters. Dwork and Naor called such a function a *pricing function* because the proposal is fundamentally an economic one: machines that currently send hundreds of thousands of spam messages each day, could, at the 10-second price, send only eight thousand. To maintain the current 2-3 billion daily messages, the spammers would require 250,000–375,000 machines.

Proof computation and verification should be performed automatically and in the background, so that the typical user e-mail experience is unchanged. The computational approach can be combined with other techniques, such as filtering. Thus, we do not propose computation at the exclusion of other techniques.

CPU-bound pricing functions suffer from a possible mismatch in processing speeds among different types of machines (desktops *vs.* servers), and in particular between old machines and the presumed new, top of the line, machines that could be used by a high-tech spam service. In order to remedy these disparities, Burrows proposed an alternative computational approach, first explored in [8], based on memory latency. His creative suggestion is to design a pricing function requiring a moderately large number of scattered memory accesses. Since memory latencies vary much less across machines than do clock speeds, memory-bound functions should prove more equitable than CPU-bound functions.

**Our Contributions.** In the current paper we explore Burrows' suggestion. After reviewing the computational approach (Section 2) and formalizing the problem (Section 3), we note that the known time/space tradeoffs for inverting one-way functions [23, 16] (where space now refers to cache) constrain the functions proposed in [8] (Section 4). We propose an abstract function, using random oracles, and give a lower bound on the amortized complexity of computing an acceptable proof of effort (Section 5)[1]. We suggest a very efficient concrete implementation of the abstract function, inspired by the RC4 stream cipher (Section 6). We present experimental results showing that our concrete memory-bound function is only about four times slower on a 233 MHz settop box than on 3.06 GHz workstation, and, interestingly, that speedup of the function is limited even if an adversary knows the access sequence and uses optimal off-line cache replacement (Section 7). Finally, we modify our concrete proposal to free the receiver from having to make memory accesses, with the goal of allowing small-memory devices to be protected by our computational anti-spam protocol.

## 2 Review of the Computational Approach

In order to send a message $m$, software operating on behalf of the sender computes a *proof of computational effort* $z = f(m, sender, receiver, date)$ for a moderately hard to compute

[1]None of [13, 20, 11, 8] obtains a lower bound.

"pricing" function $f$. The message $m$ is transmitted together with the other arguments to $f$ and the resulting proof of effort $z$[2]. Software operating on behalf of the receiver checks that the proof of effort has been properly computed; a missing proof can result in some user-prespecified action, such as placing the message in a special folder, marking it as spam, subjecting it to further filtering, and so on. Proof computation and verification should be performed automatically and in the background, so that the typical user e-mail experience is unchanged.

The function $f$ is chosen so that

1. $f$ is not amenable to amortization; in particular, computing $f(m, sender, Alice, date)$ does not help in computing $f(m, sender, Bob, date)$. This is key to fighting spam: the function must be recomputed for each recipient (and for any other change of parameters).

2. There is a "hardness" parameter to vary the cost of computing $f$, allowing it to grow as necessary to accommodate Moore's Law.

3. There is an important difference in the costs of computing $f$ and of checking $f$: the cost of sending a message should grow much more quickly as a function of the hardness parameter than the cost of checking that a proof of effort is correct. This allows us to keep verification very cheap, ensuring that the ability to wage a denial of service attack against a receiver is not exacerbated by the spam-fighting tool. In addition, if verification is sufficiently cheap, then it can be carried out by the receiver's mail (SMTP) server.

**Remark 1** *With the right architecture, the computational approach permits* single-pass send-and-forget *e-mail: To the sender, single-pass sender-and-forget means that the standard e-mail experience – including the routing of the e-mail message – is unchanged: the sender need only compose the message, perform the computation, and mail the results; it is not necessary to have round-trip communication with a third-party server (which may or may not be accessible); it is not necessary to have round-trip communication with the receiver or a server acting on the receiver's behalf. When the sender sends the message, he knows it will be received if the receiver checks her mail (at least, to the extent that this is known in e-mail today). To the receiver, single-pass send-and-forget means that once the message has arrived the proof of effort can be checked immediately, with a local computation, and the message can be handled accordingly, again without contacting a server which may or may not currently be accessible, and without further contact with the sender. Nothing needs to be stored pending (possible) future action on the part of the sender. To the Internet, single-pass send-and-forget means no additional load on the system.*

*In other words, single-pass send-and-forget means that e-mail, the killer application of the Internet, is minimally disturbed.*

**Remark 2** *We briefly remark on our use of the date as an argument to the pricing function. The receiver temporarily stores valid proofs of effort. The date is used to control the amount of storage needed. When a new proof of effort, together with its parameters, is received, one first checks the date: if the date is, say, over a week old, then the proof is rejected. Otherwise, the*

[2]Having $m$ as an argument to the function introduces some practical difficulties in real mail systems. One can instead use the following three arguments: receiver's e-mail address, date, and a nonce. However, the intuition is more clear if we include the message.

*receiver checks the saved proofs of effort to see if the newly received proof is among them. If so, then the receiver rejects the message as a duplicate. Otherwise, the proof is checked for validity.*

In [13], $f$ is a forged signature in a careful weakening of the Fiat-Shamir signature scheme. Back's proposal, called *HashCash*, is based on finding hash collisions. It is currently used to control access to bulletin boards [20]; verification is particularly simple in this scheme.

## 3 Computational Model and Statement of the Problem

The focus on memory-bound functions requires specification of certain details of a computational model not common in the theory literature. For example, in real contemporary hardware there is (at least) two kinds of space: ordinary memory (the vast majority) and *cache* – a small amount of storage on the same integrated circuit chip as the central processing unit[3]. Cache can be accessed roughly 100 times more quickly than ordinary memory, so the computational model needs to differentiate accordingly. In addition, when a desired value is not in cache (a *cache miss*), and an access to memory is made, a small block of adjacent words (a *cache line*), is brought into the cache simultaneously. So in some sense values nearby the desired one are brought into cache "for free". Our model is an abstraction that reflects these considerations, among others.

When arguing the security of a cryptographic scheme one must specify two things: the power of the adversary and what it means for the adversary to have succeeded in breaking the scheme. In our case defining the adversary's power is tricky, since we have to consider many possible architectures. Nevertheless, for concreteness we assume the adversary is limited to a "standard architecture" as follows:

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 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*), the contents of the block containing that location must be brought into the cache – a *fetch*; since every cache miss results in a fetch, we use these terms interchangeably;
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]In fact, there are multiple levels of cache; Level 1 is on the chip.

Thus, the challenge is to design a pricing function $f$ as described in Section 2, together with algorithms for computing and checking $f$, in which the costs of the algorithms are measured in terms of memory fetches and the "real" time to compute $f$ on currently available hardware is, say, about 10 seconds (in fact, $f$ may be parameterized, and the parameters tuned to obtain a wide range of target computation times).

The adversary's goal is to maximize its production of (message, proof of computational effort) pairs while minimizing the number of cache misses incurred. The adversary is considered to have won if it has a strategy that produces many (message, proof) pairs with an *amortized* number of fetches (per message plus proof) which is substantially less than the expected number of fetches for a single computation obtained in the analysis of the algorithm. We do not care if the messages are sensical or not.

We remark that it may be possible to defeat a memory-bound function with specific parameters by building a special-purpose architecture, such as a processor with a huge, fast, on-chip cache. However, since the computational approach to fighting spam is essentially an economic one, it is important to consider the cost of designing and building the new architecture. These issues are beyond the scope of this paper.

## 4 Simple Suggestions and Small-Space Cryptanalyses

In this section we will assume for simplicity that we are working in a challenge-response setting. Typically, the challenges can be derived from the messages, but we will not concern ourselves with this issue here. All the candidates discussed here seem to enjoy moderate hardness with respect to total number of operations. The difficulty lies in forcing these operations to be (random) memory accesses. Some of these ideas were considered by the authors of [8] ([2]). Unless otherwise noted, the cryptanalyses are our own.

In the sequel, we follow [8] in assuming the sender has up to 8MB of cache, as the SUN Ultra 60 is such a machine. Thus, the goal is to design a function for a machine that has, say, at least 16MB of memory. Machines with less memory are effectively not supported with this approach.

### 4.1 Meet in the Middle

In the meet-in-the-middle proposal, the legitimate sender solves a partially specified instance of double encryption:
*Input. Two 64-bit strings $x$ and $y$, and $56 - k$ bits for each of two keys $K_1$ and $K_2$ ($k$ is a hardness parameter).*
*Output. 56-bit keys $K_1$ and $K_2$ such that $E_{K_2}(E_{K_1}(x)) = y$. Here, $E$ is any keyed encryption scheme, such as DES.*
The best technique known for this problem is the famous meet-in-the-middle attack on double block ciphers, described in [12, 19, 32]. The legitimate e-mail sender carries out this "attack". Let $S_i$ be the set of all possible choices for key $K_i$ consistent with the bits given in the challenge. Suppose the attacker stores $E_{K_1}(x)$ for all keys $K_1 \in S_1$ in a table $T$, so $|T| = |S_1|$. For each key $K_2 \in S_2$, the sender computes $E^{-1}_{K_2}(y)$ and checks whether or not the result is in the set of stored values. If so, then the challenge has been resolved.

If the table $T$ fits into cache, then the challenge may be resolved completely without cache misses. Choosing $k$ so that the table $T$ is, say, twice the size of the largest cache, one might hope it may be possible to force many cache misses. However, there are several (related) reasons why this doesn't work.

(1) For cache size $c$, the attack can be carried out with no cache misses in time proportional to $(|S_1|/c)\cdot|S_2|$. This is done by conceptually dividing $T$ into blocks of size $c$ and then repeatedly computing and caching a block of $T$ and, for each $K_2 \in S_2$, checking whether $E_{K_2}^{-1}(y)$ is in the cached block. (2) Even if one computes the entire table $T$ and then loads into cache a single cache-sized block of $T$, there is non-negligible chance that the challenge will be resolved *in cache*, without random accesses to memory. For example, if the cache has size one quarter the size of $T$, once the cache has been loaded there is a 1/4 chance that resolution will occur without further cache misses. (3) Sequentiality/Cache-line attacks[4]: the attacker computes $E_{K_2}^{-1}(y)$ for many values of $K_2$, until so many values to look up in $T$ have been computed that it makes sense to *sequentially* read into cache a large block of $T$. By exploiting locality the constraining cost becomes memory throughput rather than memory latency.

## 4.2 Subset Sum

Here the legitimate sender of a message solves an instance of subset sum:
*Input. Random $2k$-bit numbers $a_1, \ldots, a_{2k}$ and integer target $T$.*
*Output. Subset $S \subseteq \{1, \ldots, 2k\}$ such that $\sum_{i \in S} a_i = T \bmod 2^{2k}$.*
There may be several reasons for optimism (at least, before one realizes that meet-in-the-middle is not so good). Two competing techniques for solving subset sum are dynamic programming and the LLL algorithm. The LLL algorithm is probably too expensive[5]. In the naive dynamic programming approach, the numbers are partitioned into two sets of size $k$, and all $2^k$ subset sums of each partition are computed. The pairs of sums (one from each partitioned) are then considered in sorted order. This appears to lead to an $O(2^k)$ time and $O(2^k)$ space solution (this approach works for any group). However, Schroeppel and Shamir obtained a $T = O(2^k)$, $S = O(2^{k/2})$ subset sum algorithm (with $2k$ inputs) [33]. In our setting, $k \approx 22$, so the required space is $2^{11}$ times a constant hidden by the "O" notation, so the program is likely to fit in a reasonably sized cache (details omitted from this extended abstract).

## 4.3 Easy-to-Compute Functions

We now consider the functions proposed in [8]. These are essentially iterates of a single basic "random-looking" function $g$. They vary in their choice of basic function. The basic function has the property that a single function inversion is more expensive than a memory look-up.

Let $n$ and $\ell$ be parameters and let $g : \{0,1\}^n \longrightarrow \{0,1\}^n$. Let $g_0$ be the identity function and for $i = 1 \ldots \ell$, let the function $g_i(x) = g(g_{i-1}(x)) \oplus i$.
*Input. $y = g_\ell(x)$ for some $x \in \{0,1\}^n$ and $\alpha$, a hash of the string of values $x, g_1(x), \ldots, g_\ell(x)$.*
*Output. $x' \in g_\ell^{-1}(y)$ such that the string $x', g_1(x'), \ldots, g_\ell(x')$ hashes to $\alpha$.*
The hope is that the best way to resolve the challenge is to build a table for $g^{-1}$ and to work

[4]This general class of attacks, not specific to meet-in-the-middle, was pointed out to us by Martin Abadi and Mike Burrows.

[5]If $m$ is the maximum number of bits in a coefficient of the original lattice vectors, $n$ is the dimension of the space, and $q$ is the dimension of the lattice, then the algorithm runs in time $O(nq^5m^3)$ [24].

backwards from $y$, exploring the tree of pre-images[6]. Since forward computation of $g$ is assumed to be quite easy, constructing the inverse table should require very little total time compared to the memory accesses needed to carry out the proof of effort.

The number of possible preimages for an element in the range of $g_\ell$ is fairly large (on the order of $\ell$). Intuitively, the sender must search through many of these possible preimages in order to find one that yields a path that hashes to $\alpha$. The total size of the tree of pre-images is on the order of $\Theta(\ell^2)$, so in the best case (for the memory-bound approach) the required number of cache misses would also be $\Theta(\ell^2)$. In contrast, verification would require *no* memory access, and $\ell$ forward computations of $g$.

The limitation of this approach is that, since $g$ can be computed (in the forward direction) with no memory accesses, there is a time/space tradeoff for *inverting* $g$ in which no memory accesses are performed. General time-memory tradeoffs to invert one-way functions were first explored by Hellman [23], and the approach was made more rigorous by Fiat and Naor [16]. Let $\mathbf{T}$ be the time it will take the scheme invert $g$, and let $\mathbf{S}$ be the memory available to carry out the inversion. In the setting of memory-bound functions, the amount of memory $\mathbf{S}$ would be the size of the cache (we are trying to prove we don't need to go to main memory, so we are interested in what can be done when the only space available for the computation is the cache).

Let $\text{cpu}_g$ be the number of CPU cycles needed to carry out a forward computation of $g$: that is, the computation $g(x)$ requires at most $\text{cpu}_g$ CPU cycles. Let $N = 2^n$ be the size of the domain of $g$. The tradeoff is:

$$\mathbf{T} \cdot \mathbf{S}^2 = O(N^2)\text{cpu}_g$$

after preprocessing (which is purely CPU-bound)[7].

For the parameters in [8] we have, roughly, $N = S/2$, so for those functions an inversion can be computed with about four forward computations of $g$, without *no* memory fetches. Thus, in order to make table lookup "preferable" to inversion via the trade-off, $\text{cpu}_g$ cannot be too small: otherwise, it would be faster to circumvent the table lookups, and resolving the computational challenge would become CPU-bound.

As clock speeds increase, it will be necessary to modify $g$ so that it remains faster to invert $g$ via table lookups than via the time-space tradeoff. Thus, the class of functions proposed in [8] is ultimately CPU-bound as well. However, the structure of these functions – the fact that challenges can be resolved using scattered memory accesses instead of CPU cycles – dampens the effect of Moore's Law: the memory-intensive solution will require less real time than the CPU-intensive solution until CPUs become very fast. Until that point, the functions of [8] are more egalitarian for the senders than purely CPU-intensive functions.

Finally, we note (1) verification costs rise exponentially as the CPU cost of the function $\text{cpu}_g$ is altered to keep pace with Moore's Law and (2) the time/space tradeoff in [23, 16] is not tight, and any improvement to the tradeoff will mandate an increase in $\text{cpu}_g$. Indeed, a recent result of Oechslin improves the tradeoff by a factor of two [31].

The analysis of easy-to-compute functions suggests basing computational challenges on functions that are (in some sense) *hard in both directions.*

[6]The root of the tree is labelled with $y$. A vertex at distance $d \geq 0$ from the root having label $z \in \mathit{Range}(g_{\ell-d})$ has one child labeled with each $z' \in g^{-1}(z \oplus (\ell - d)) \in \mathit{Range}(g_{\ell-d-1})$.

[7]This is the cost of finding a single inverse. Also, the construction in [16] ensures probabilistic domain coverage. The construction is easily modified to enable the finding of *all* preimages, and to ensure that the domain is completely covered.

## 5 An Abstract Function and Lower Bound on Cache Misses

In this section we describe an "abstract" pricing function and prove a tight lower bound on the number of memory accesses that must be made in order to produce a message acceptable to the receiver, in the model defined in Section 3. The function is "abstract" in that it uses idealized hash functions, also known as random oracles. A concrete implementation is proposed in Section 6.

**Meaning of the Model and the Abstraction:** Our computational model implicitly constrains the adversary by constraining the architecture available to the adversary. Our use of random oracles for the lower bound argument similarly constrains the adversary, as there are some things it cannot compute without accessing the oracles. We see two advantages in such modelling: (i) It provides rationale to the design of algorithms such as those of Section 6, this is somewhat similar to what Luby and Rackoff [26] did for the application of Feistel Permutations in the design of DES; (ii) If there is an attack on the simplified instantiation of the algorithm of Section 6, then the model provides guidelines for modifications. Note that we assume that the arguments to the random oracle must be in cache in order to make the oracle call.

The inversion techniques of [23, 16] do not apply to truly random functions, as these have large Kolmogorov complexity (no short representation). Accordingly, our function involves a large *fixed forever* table $T$ of truly random $w$-bit integers[8]. The table should be approximately twice as large as the largest existing caches, and will dominate the space needs of our memory-bound function.

We want to force the legitimate sender of a message to take a random walk "through $T$," that is, to make a series of random accesses to $T$, each subsequent location determined, in part, by the contents of the current location.

Such a walk is called a *path.* The algorithm forces the sender to explore many different paths until a path with certain desired characteristics is found. We call this a *successful* path. 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. Verification requires work proportional to the path length, determined by a parameter $\ell$. Each path exploration is called a *trial.* The expected number of trials to find a successful path is $2^e$, where $e$ (for "effort") is a parameter. The expected amount of work performed by the sender is proportional to $2^e$ times the path length.

### 5.1 Description of the Abstract Algorithm

The algorithm uses a modifiable array $A$, initialized for each trial, of size $|A|w > b$ bits (recall that $b$ is the number of bits in a memory block, or cache line)[9].

Before we present the abstract algorithm, we introduce a few hash functions $H_0, H_1, H_2, H_3$, of varying domains and ranges, that we model as idealized random functions (random oracles). The function $H_0$ is only used during initialization of a path. It takes as input a message $m$,

[8]"Fixed forever" means fixed until new machines have bigger caches, in which case the function must be updated.

[9]The intuition for requiring $|A|w > b$ is that, since $A$ cannot fit into a single memory block, it is more expensive to fetch $A$ into cache than it is to fetch an element of $T$ into cache.

sender's name (or address) $S$, receiver's name (or address) $R$, and date $d$, together with a trial number $k$, and produces an array $A$. The function $H_1$ takes an array $A$ as input and produces an index $c$ into the table $T$. The function $H_2$ takes as input an array $A$ and an element of $T$ and produces a new array, which gets assigned to $A$. Finally, the function $H_3$ is applied to an array $A$ to produce a string of $4w$ 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.

The path in a generic trial is given by:

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

Path exploration is repeated for $k = 1, 2, \ldots$ until success occurs. The information for identifying the successful path is simply all five parameters and the final $H_3(A)$ obtained during the successful trial[10].

Verification that the path is indeed successful is trivial: the verifier simply carries out the exploration of the one path and checks that success indeed occurs with the given parameters and that the reported hash value $H_3(A)$ is correct.

The connection to Algorithm MBound, described in Section 6, will be clear: need only specify the four hash functions. To keep computation costs low in MBound, we will not invoke full-strength cryptographic functions in place of the random oracles, nor will we even modify all entries of array $A$ at each step.

The size of $A$ also needs consideration. If $A$ is too small, say, a pointer into $T$, then the spammer can mount an attack in which many different paths (trials for either the same or different messages) can be explored at a low amortized cost, as we now informally describe. At any point, the spammer can have many different $A$'s (that is, $A$'s for different trials) in the cache. The spammer then fetches a memory block containing several elements of $T$, and advances along each path for which some element in the given memory block was needed. This allows exploitation of locality in $T$. Thus, intuitively, we should choose $|A|$ sufficiently large that it is infeasible to store many different $A$'s in the cache.

## 5.2 Lower Bound on Cache Misses

We now prove a lower bound on the amortized number of block transfers that any adversary constrained as described in Section 3 must incur in order to find a successful path. Specifically, we show that the *amortized* complexity (measured in the number of memory fetches per message) of the abstract algorithm is asymptotically tight.

The computation on each message must follow a specific sequence of oracle calls in order to make progress. The adversary may make any oracle calls it likes; however, to make progress on a path it must make the specified calls. *By watching an execution unfold, we can observe*

[10]The value of $H_3(A)$ is added to prevent the spammer from simply guessing $k$, which has probability $1/2^e$ of success.

*when paths begin, and when they make progress.* Calls to the oracle that make progress (as determined by the history) are called *progress calls.*

A path *begins* when a call $H_0(m, S, R, d, k)$ is first made. The path is completely identified by the five parameters to $H_0$.

**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 3. Under the following additional conditions, the expectation, over choice of $T$ and $H$ and coin flips of $\mathcal{A}$, of the amortized complexity of generating a proof of effort that will be accepted by a verifier is $\Omega(2^e \ell)$:*

- $|T| \geq 2s$ *(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 $2^{o(w)}2^e\ell$.*
- *$\ell$ is large enough so that the spammer cannot call the oracle $2^\ell$ times.*

**Remark 3** *First note that $|A|$ is taken to be much larger than $b/w$. We already noted that if $|A|$ is very small than a serious attack is possible. However, even if $|A|$ is roughly $b/w$, it is possible to attack the algorithm by storing many copies of $T$ under various permutations. In this case the adversary can hope to concurrently be exploring about* $\log s$ *paths for which a single memory block contains the value in $T$ needed by all* $\log s$ *paths. Hence, if (for some reason) it is important that $|A| \leq O(b/w)$ we can only get a lower bound of the form $\Omega(2^e\ell/\log s)$.*

**Remark 4** *The theorem also holds if we replace* expected *amortized cost (over $T, H$, and flips) with "with high probability".*

*Proof* (of Theorem 1): We start with an easy lemma regarding the number of oracle calls needed to find a successful path.

**Lemma 1** *The 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 following lemma is a completely elementary preliminary used in our lower bound proof.

**Lemma 2** *Let $b_1 \ldots b_m$ be independent unbiased random bits and let $k \leq m$. Suppose we have a system that, given a hint of length $B < k$ (which may be based on the value of $b_1 \ldots b_m$), produces a subset $S$ 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 guessing system.*

*Proof:* Each hint yields an assignment to the indicated bits. The probability, over choice of $b_1, \ldots, b_m$, that the assignment is consistent with the values of the elements of $S$ is $2^{-k}$. Thus on average each hint yields the correct answer with probability $2^{B-k}$. $\diamondsuit$

We now get to the main content of the lower bound and to the key lemma (Lemma 3): We break the execution into intervals in which, we argue, 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[11].

We first motivate our definition of an interval. We want to think of each $A$ as incompressible, since it is the output of a random function. However, if, say, this is the beginning of a path eploration, and $A = H_0(m, S, R, d, k)$, then it may require less space simply to list the arguments to $H_0$; since our model does not charge (much) for oracle calls, the adversary incurs no penalty for this. For this reason, we will focus 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. Our definition of an interval will allow us to focus on progress on paths with "mature" $A$'s.

Let $n = s/|A|$; it is helpful to think of $n$ as the number of $A$'s that can simultaneously fit into cache (assuming they are incompressible). A progress call is *mature* if it is the $j$th progress call of the path, for $j > \ell/2$ (recall that $\ell$ is the length of a path). 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 $8n$ mature progress calls (spread over any number of paths) have been made to oracle $H_1$.

**Lemma 3** *The average number of memory accesses made during an interval is $\Omega(n)$, where the average is taken over the choice of $T$, the responses of the random oracles, and the random choices made by the adversary.*

Note that between intervals we allow the adversary 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 1, 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 $8n$, which is $\Omega(2^e\ell/n)$. (Note that we have made no attempt to optimize the constants involved.)

*Proof:* (of Lemma 3) Intuitively, 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* $8n$ 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 $H_2$. Let $\beta$ be the average number of blocks sent by the main memory to the cache during an interval, and we assume for the sake of contradiction that $\beta = o(n)$ (the lemma

[11]In fact, our proof will hold 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 state is reduced to $sw$ bits once again.

asserts that $\beta = \Omega(n)$). We know that 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 $\beta b$ bits altogether from the memory. The key to the lemma is, intuitively, that the relatively few possibilities in requesting blocks by the cache imply that many different elements of $T$ indicated by the indices returned by the $8n$ mature calls to $H_1$ have to be stored in the same set of blocks. We will argue that this implies that a larger than $s$ part of $T$ can be reconstructed from the cache contents alone, which is a contradiction given the randomness of $T$.

We now proceed more formally. Lemma 3 will follow from a sequence of claims. The first is that there are many entries of $T$ for which many possible values are consistent with the cache contents at the beginning of the interval. That is, $T$ is largely unexplored from the cache's point of view.

**Claim 5.1** *There exist $\gamma, \delta \geq 1/2$ such that: given the cache contents at the beginning of the interval, it is expected that there exists a subset of the entries of $T$, called $T'$, of size at least $\delta|T|$ such that for each entry $i$ in $T'$ there is a set $S_i$ of $2^{\gamma w}$ possible values for $T[i]$ and all the $S_i$'s are mutually consistent with the cache contents.*

*Proof:* We want to apply Sauer's Lemma to show that, given the cache content, at the start of the interval, there are many words with high entropy, *i.e.*, with lots of possibilities. We know that the cache size is $s$ words of width $w$, or $sw$ bits total. Consider an assignment to $T$ as a binary vector of length $sw$.

The expected number of assignments to $T$ consistent with a given cache content is $2^{(|T|-s)w}$. Each of these is described by a binary vector of length $|T|w$. Sauer's Lemma (described in Alon and Spencer [9]) says that in such a large collection of vectors there must be $d$ variables (bit positions) that appear in all $2^d$ combinations, where $d$ satisfies $\binom{|T|w}{(d+1)} \geq 2^{(|T|-s)w}$ (more precisely $\sum_{i=0}^{d} \binom{|T|w}{i} \geq 2^{(|T|-s)w}$.) Thus, if $s = |T|/2$ then we have that $\binom{|T|w}{(d+1)} \geq 2^{|T|w/2}$. Since $\binom{|T|w}{(d+1)} \leq (\frac{e|T|w}{d})^d$ we have that $d \log(e|T|w/d) \geq |T|w/2$ and $d \geq 1/2|T|w$. So for some $T'$ of size $1/2$ times $|T|$ there are $|T'|$ entries in $T$ where the number of possibilities to the entry is at least $2^{w/2}$ (*i.e.*, $\gamma \geq 1/2$). In the sequel, we will concentrate solely on these $|T'|$ entries. If there is a mature progress call with an element among these entries, we call this an *element to be learned.* $\diamondsuit$

From now on we assume that we have cache content consistent with a large number of possibilities for $T'$ as in the claim and use this cache configuration to show that it is possible to extract many entries of $T'$.

**Claim 5.2** *If the number $\beta$ of memory accesses is $o(n)$, then the number of different paths on which a mature progress call is made during an interval is at most $3n$.*

*Proof:* When a mature progress call is made with a value $A$, this $A$ can come from three sources: 1) it was stored in the cache at the start of the start of the interval; 2) it was developed from a previous call to $H_2$ where the value of $A$ for that call was itself stored in the cache; 3) it was developed "from scratch" *i.e.*, all (at least $\ell/2$) points on the path are computed during the interval. Given an execution it is easy to classify each progress call into one of these three types. Consider first all the calls of types (1) and (2), that is, calls where at least one point on the path is not accessed in $H_1$ in the interval. Suppose that there are at least $2n$ of them, so that at on at least $2n$ different paths calls of types (1) or (2) are made. Let's examine the first calls to $H_1$ in on all the paths for which the preceding call to $H_2$ is not present. Together, the inputs to these calls require at least $2n|A|w$ bits of information. Since the cache holds only $n|A|w$ bits and the $A$'s are incompressible (from the randomness of $H_2$), for the algorithm to have non-negligible probability of being correct it must obtain at least $n|A|w$ bits from memory during the interval. Since $|A|w > b$ (where $b$ is the block size in bits), this means at least $\Omega(n)$ memory accesses, contradicting the assumption that $\beta = o(n)$.

Consider now type (3) progress calls, *i.e.*, calls on paths that are explored from scratch during the interval. Since a mature progress call requires that $\ell/2$ calls have been made to $H_2$ (in bringing the path to maturity), it follows that the values stored in the locations of $T$ appearing as arguments in these calls must be known during the interval. However, we argued in Claim 5.1 that a large part (in fact, half) of $T$, called $T'$, is missing almost fully from the cache. Suppose there are $n$ paths of type (3), requiring together $n\ell/2$ accesses to $T$. There are two possibilities: Either the total number of elements of $T'$ accessed in these $n\ell/2$ accesses to $T$ exceeds $|T'|/2$, or not. If this number exceeds $|T'|/2$, then we need to retrieve $\Omega(|T'|w\gamma/2b)$ (which is greater than $\Omega(n)$) blocks of memory (by Information Theory). If this number is less than $|T'|/2$, then we claim that the adversary has witnessed an unlikely event: All the elements accessed in the length $\ell/2$ prefixes of the $n$ paths fall into a set of size at most $|T| - |T'| + |T'|/2 \leq 3|T|/4$ (by Claim 5.1). Fix such a set. The probability of this event is $(3/4)^{n\ell/2}$. There are at most $2^{|T|}$ such sets. Suppose that the spammer has examined $z$ paths. Then the probability that in these $z$ paths there is such an $n$-collection of paths is multiplied by $\binom{z}{n}$. Given that we assumed that $2^\ell$ is not a feasible number for the adversary we know that $\ell n >> |T|$ and we get that $\binom{z}{n} \cdot 2^{|T|}$ is $\ll (4/3)^{n\ell/2}$. So with high probability the spammer cannot find $n$ such paths.

So we have at most $2n$ paths of types (1) and (2) combined, and at most $n$ paths of type (3), for a total of at most $3n$ paths. ◇

It therefore follows that in a typical interval there are at least $8n - 3n = 5n$ pairs of consecutive mature progress calls to $H_1$ on a common path. Thus, for example, one path may experience $5n + 1$ mature progress calls, or each of $n$ paths may experience at least 6 mature progress calls, or something in between. Each such pair of calls to $H_1$ is separated by a call to $H_2$ which requires the contents of the location of $T$ specified by the first $H_1$ call in the pair. It is these interstitial calls to $H_2$ that are of interest to us: because their preceding calls to $H_1$ first occur during the interval, and $H_1$ is random, it cannot be known at the start of the interval which elements of $T$ will be needed as arguments to these calls to $H_2$. Intuitively, the adversary *must* go to main memory to find an expected $(|T| - s)/|T| > 1/2$ of them.

**Claim 5.3** *In a typical interval there are at least* $5n$ *pairs of consecutive mature progress calls to* $H_1$ *on a common path. Thus there are also* $5n$ *pairs of calls to* $H_1$ *and then to* $H_2$ *on the same path.*

**Good Tuples.** Consider the set of $8n$-tuples over $\{1, \ldots, |T|\}$ as the set of possible answers $H_1$ returns on the mature progress calls in the interval; there are $|T|^{8n}$ such tuples. Fix all other random choices: the value of $T$, the previous calls to $H_1$ and $H_2$ and the random tape of the spammer). The spammer's behavior in an interval is now determined solely by this $8n$-tuple.

If the spammer can defeat our algorithm, then, by Markov's inequality, for at least half the $8n$-tuples, the spammer completes the interval retrieving at most $2\beta$ blocks. Call these $8n$-tuples the *good* ones, and denote by $G$ the set of good tuples.

We first claim that in most of the tuples in $G$, the spammer goes frequently into $H_2$ with values $T[i]$ where $i \in T'$.

**Claim 5.4** *Let $T'$ be any subset of the entries of $T$ of size at least $\delta|T|$. Consider the set $G$ of good $8n$-tuples over $\{1, \ldots, |T|\}$ as the set of possible answers to queries to $H_1$ during the interval. Then, except for at most an exponential in $n$ fraction of $G$, the spammer must use an entry in $T'$ for a call to $H_2$ at least $n$ times during an interval.*

*Proof:* Since $G$ is at least half the set of all possible $8n$-tuples, it suffices to prove the claim for the set of all $8n$-tuples. By definition, an interval contains $8n$ mature progress calls to $H_1$, and we argued in Claim 5.2 that these occur in at most $3n$ paths. When the spammer calls $H_1$ and gets a value $c$, there is probability at least $\delta = 1/2$ that this $c$ is in $T'$. The spammer may decide to pursue this $c$ (that is, to try to learn $T[c]$) in this interval or not. If he decides not to pursue it, then the current path will make no further progress in the interval. This can happen at most $3n$ times (since there are at most $3n$ paths of interest). Therefore, if $(\delta 8n) - 3n \geq n$ then we get that at least $n$ (not necessarily distinct) values of $T'$ have to be retrieved from main memory during the interval for the subsequent call to $H_2$. $\diamondsuit$

Since the memory size is $m$, there will be many good $8n$-tuples that share the same set of blocks; that is, by retrieving one set of blocks all elements appearing in many good $8n$-tuples can be reconstructed in the cache. More precisely, a $1/\binom{m}{2\beta}$ fraction of the good tuples share the same set of blocks. Let $G' \subset G$ be the largest such set of $8n$-tuples.

**Too Many Bits From Too Few.** The idea for deriving the contradiction is to show

1. If there exists a "small" number of tuples in $G'$ whose union covers a "large" portion of $T'$, then the memory can transmit "too many bits" of $T'$ using "too few" bits of communication (Claim 5.5); and
2. If $\beta = o(n)$ then indeed there exists a "small" number of tuples in $G'$ whose union covers a "large" portion of $T'$.

**Claim 5.5** *If there exist $2x/n$ $8n$-tuples in $G'$ whose union covers $x$ elements of $T'$, then memory can transmit the missing $\gamma w$ values of each of $x$ entries of $T'$ by sending the $2\beta b$ bits describing the content of the common blocks and in addition sending, for each of the $2x/n$ tuples in the cover,*

1. *the $8n \log |T|$ bits specifying the $8n$-tuple and*
2. *information specifying which calls to $H_2$ in the execution have the correct parameters (there may be some "bogus" calls to $H_2$ in which the wrong values for elements of $T$ are used as parameters). If the interval contains $z$ calls to $H_2$ then this requires $\log \binom{z}{n/2}$ bits, which is less than $n/2 \log z$.*

*So altogether, multiplying by $2x/n$ we get that it suffices for $2\beta b + 16x \log |T| + x \log z$ bits to be sent from the memory to the cache in order for the cache to learn $\gamma w$ bits for each of $x$ entries in $T'$, or $x\gamma w$ bits altogether.*

Recall that $\gamma \geq 1/2$, so we are comparing $x\gamma w \geq xw/2$ (the number of bits learned) to $2\beta b + 16x \log |T| + x \log z$ (the number of bits transmitted). Since $w$ was taken to be much larger than $\log |T|$ and $2^{w/2}$ much larger than the amortized number of oracle calls per interval, $\log z$ is much smaller than $w$ and we only have to worry about the $2\beta b$ term.

Assume that $\beta \leq n/80 = s/(80|A|)$ and, for simplicity, that $m$, the memory size in blocks, is $|T|^2$ (recall that in our model $m$ is polynomial in $s$, and in our theorem $|T| = \Theta(s)$). Set $x = 4\beta b/w$. Recall that $G'$ is the largest set of good tuples agreeing with a set of $2\beta$ blocks, *i.e.*, consisting of at least a $1/\binom{T^2}{2\beta}$ fraction of the good tuples. We will show (Claim 5.8) that $G'$ has $2x/n$ $8n$-tuples whose union is of size at least $x$, that is, the premise of Claim 5.5 holds (this will be sufficient for a contradiction).

**Claim 5.6** *Suppose that we have a collection of $8n$-tuples and we want to cover at least $x$ values in $T'$ using only $2x/n$ members of the collection (assume that the collection is at least that large). If this is impossible then there is a set $X \subset T'$ of size $x$ such that* every *member of the collection has at least $n/2$ entries in $X$.*

*Proof:* We prove the contrapositive. If there is no set $X \subset T'$ of size $x$ such that *every* member of the collection has at least $n/2$ entries in $X$, then we can build a large coverage in a greedy manner, each time adding at least $n/2$ new entries in $T'$. At any point during the process the union $X$ of all tuples we added should be of size less than $x$ and hence there should be a tuple in the collection with $n/2$ entries in $T'$ but not in $X$. So after $2x/n$ steps we have covered $x$ entries of $T'$. $\diamondsuit$

**Claim 5.7** *Suppose that we have subset $X$ of size $x$ of entries in $T'$. Then the probability that a random $8n$-tuple contains more than $n/2$ entries in $X$ is at most $(2^8 x/|T|)^{n/2}$.*

*Proof:* This is by simple computation: for a fixed set of $n/2$ entries out of the $8n$ this probability is $(x/|T|)^{n/2}$ and there are at most $\binom{8n}{n/2} \leq 2^{8n/2}$ such subsets. $\diamondsuit$

**Claim 5.8** *$G'$ contains $2x/n$ $8n$-tuples whose union is of size at least $x$.*

*Proof:* Suppose that this is not the case and the $2x/n$ tuples covering $x$ elements do not exist. Then as we have seen above in Claim 5.6 there is a set $X$ of size $x$ where each tuple in $G'$ has at least $n/2$ entries in $X$. Let $C_X$ denote the set of all $8n$-tuples containing at least $n/2$ entries in $X$. We know from Claim 5.7 that $C_X$ contains at most a $(2^8 x/|T|)^{n/2}$ fraction of the set of all $8n$-tuples. Since $|G'|/|G| \geq 1/\binom{T^2}{2\beta}$ and $G$ contains at least half the $8n$-tuples, $G'$ consists of at least a $1/2\binom{T^2}{2\beta}$ fraction of all $8n$-tuples.

So we need to compare these fractions. That is, if

$$\frac{1}{2}\binom{|T|^2}{2\beta}^{-1} > \left(\frac{2^8 x}{|T|}\right)^{n/2} \tag{1}$$

then we have a contradiction: we know that $G' \not\subseteq C_X$, *i.e.*, $G'$ is too large to be compressed into $X$.

To show Equation 1 it suffices to show that

$$\frac{|T|^{n/2}}{2\binom{|T|^2}{2\beta}} > (2^8x)^{n/2}.$$

Now,

$$\frac{|T|^{n/2}}{2\binom{|T|^2}{2\beta}} > \frac{|T|^2}{2|T|^{4\beta}} = \frac{1}{2}|T|^{n/2-4\beta}.$$

This it suffices to show that

$$\frac{1}{2}|T|^{n/2-4\beta} > (2^8x)^{n/2}.$$

Taking logs and then dividing by $n/2$, we see that we need to compare $8+\log x$ and

$$\frac{n-8\beta}{n}\log|T| - 2/n \geq \frac{s/|A| - 8s/80|A|}{s/|A|}\log|T| - 2/n = \frac{9}{10}\log|T| - 2/n.$$

But since $x = 4\beta b/w = 4sb/(80|A|w)$ and $|A| \geq s^{1/5}b/w$ we get that $x \leq 1/5s^{4/5}$ and indeed $8+\log x$, which is less than $8+\frac{4}{5}\log s$ for sufficiently large $|T|$, is smaller than $\frac{9}{10}\log|T|$. $\diamondsuit$

This concludes the proofs of Lemma 3 and Theorem 1 $\diamondsuit$

## 6 A Concrete Proposal

In this section we describe a concrete implementation of the abstract algorithm of Section 5, which we call Algorithm MBound. As in the abstract algorithm, our function involves a large *fixed forever* array $T$, now of $2^{22}$ truly random 32-bit integers[12]. In terms of the parameters of Section 5, we have $|T| = 2^{22}$ and $w = 32$. This array requires 16 MB and dominates the space needs of our memory-bound function, which requires less than 18 MB total space.[13] The algorithm requires in addition a fixed-forever truly random array $A_0$ containing 256 32-bit words. $A_0$ is used in the definition of $H_0$. Note that $A_0$ is incompressible.

### 6.1 Description of MBound

Our proposal was inspired by the (alleged) RC4 pseudo-random generator (see, *e.g.*, the descriptions of RC4 in [18, 27, 29]).

**Description of $H_0$.** Recall that we have a fixed-forever array $A_0$ of 256 truly random 32-bit words. At the start of the $k$th trial, we compute $A = H_0(m, S, R, d, k)$ by first computing (using strong cryptography) a 256-word mask and then XORing $A_0$ together with the mask. Here is one way to define $H_0$:

[12] "Fixed forever" means fixed until new machines have bigger caches, in which case the function must be updated.

[13] To send mail, a machine must be able to handle a program of this size.

1. Let $\alpha_k = h(m, S, R, d, k)$ ($|\alpha_k| = 128$), for a cryptographically strong hash function $h$ such as, say, SHA-1.
2. Let $\eta(\alpha_k)$ be the $2^{13}$-bit string obtained by *concatenating* the $2^7$-bit $\alpha_k$ with itself $2^6$ times[14]. Treating the array $A$ as a $2^{13}$-bit string (by concatenating its entries in row-major order), we let $A = A_0 \oplus \eta(\alpha_k)$. Note that, unlike in the case of RC4, our array $A$ is *not* a permutation of elements $\{1, 2, \ldots, 256\}$, and its entries are 32 bits, rather than 8 bits.

We initialize $c$, the *current* location in $T$, to be the last 22 bits of $A$ (when $A$ is viewed as a bit string). In the sequel, whenever we say $A[i]$ we mean $A[i \bmod 2^8]$; similarly, by $T[c]$ we mean $T[c \bmod 2^{22}]$.
The path in a generic trial is given by:

```
Initialize Indices:
  $i = 0$;  $j = 0$
Walk for $\ell$ steps ($\ell$ is the path length):
  $i = i + 1$
  $j = j + A[i]$
  $A[i] = A[i] + T[c]$
  $A[i] = RightCyclicShift(A[i], 11)$ (shift forces all 32 bits into play)
  $Swap(A[i], A[j])$
  $c = T[c] \oplus A[A[i] + A[j]]$
Success occurs if the last $e$ bits of h(A) are all 0.
```

In the last line, the hash function $h$ can again be SHA-1. It is applied to $A$, treated as a bit string.

The principal difference with the RC4 pseudo-random generator is in the use of $T$: bits from $T$ are fed into MBound's pseudo-random generation procedure, both in the modification of $A$ and in the updating of $c$.

In terms of the abstract function, we can tease our proposal apart to obtain, *roughly*:

**Description of $H_1$ (updates $c$, leaves $A$ unchanged).** The function $H_1$ is essentially

```
  $i = i + 1$
  $j = j + A[i]$
  $v = A[i] + T[c]$ ($v$ is a temporary variable)
  $v = RightCyclicShift(v, 11)$
  $c = T[c] \oplus A[A[j] + v]$
```

**Description of $H_2$ (updates $A$).**

```
  $A[i] = A[i] + T[c]$
  $A[i] = RightCyclicShift(A[i], 11)$
  $Swap(A[i], A[j])$
```

[14]The reason we concatenate the string in order to generate $\eta(\alpha_k)$, rather than generate a cryptographically strong string of length $2^{13}$ is to save CPU cycles - this is an operation that is done many times and if each bit of $\eta(\alpha_k)$ is strong it could make the scheme CPU bound.

**Description of $H_3$.** The hash function $H_3(A)$ is simply some cryptographically strong hash function with 128 bits of output, such as SHA-1.

This all but completes the description of Algorithm MBound and its connection to our abstract function; it remains to choose the parameters.

### 6.2 Parameters for MBound

We can define the computational puzzle solved by the sender as follows.

**Input.** A message $m$, a sender's alias $S$, a receiver's alias $R$, a time $t$, the table $T$ and the auxiliary table $A_0$.

**Output.** $m, S, R, d, i$ and $\alpha$ such that $1 \leq i \leq 2e$ and the $i$th path (that is, the path with trial number $k = i$), is successful and $\alpha$ is the result of hashing the final value of $A$ in the successful path.

If $i > 2^{2e}$, the receiver rejects the message (with overwhelming probability one of the first $2^{2e}$ trials should be successful).

To be specific in the following analysis, we make several assumptions. These assumptions are reasonable for current technology, and our analysis is sufficiently robust to tolerate substantial changes in many of these parameters. Let $P$ be the desired expected time for computing the proof of effort and let $\tau$ be the memory latency. We assume that $P$ is 10 seconds and $\tau$ is .2 microseconds. We also assume that the maximum size of the fast cache is 8 MB and that cache lines (memory blocks) are 64 bytes wide (so blocks contain $b = 512$ bits).

The output conditions ensure that for a random starting point, the probability of a successful output is $1/2^e$. The expected number of walks to be checked is $2^e$. Therefore the expected value of $P$ is

$$E[P] = 2^e \cdot \ell \cdot \tau.$$

The cost of verification by the receiver is essentially $\ell$ cache misses, by following the right path. (In Section 8 we discuss how to reduce or eliminate these cache misses.)

We have not yet set the values for $e$ and $\ell$. Choosing one of these parameters forces the value of the other one. Consider the choice of $e$: one possibility might be to make $e$ very large, and the paths short, say, even of length 1. This would make verification extremely cheap. However, while the good sender will explore the paths sequentially, a cheating sender may try several paths in parallel, hoping to exploit locality by batching several accesses to $T$, one from each of these parallel explorations. In addition, $A$ changes slowly, and to get to the point in which many "mature" values of $A$ cannot be compressed requires that many entries of $A$ have been modified. For our concrete proposal, therefore, we let $\ell = 2048$. Then $2^e = P/\ell\tau = 10/(2048 * 2 * 10^{-7}) \approx 24,414$.

## 7 Experimental Results

In this section we describe several experiments aimed at establishing practicality of our approach and verifying it experimentally. First we compare our memory-bound function performance to that of the CPU-intensive HashCash function [20] on a variety of computer architectures. We confirm that the memory-bound function performance is significantly more platform-independent. We also measure the solution-to-verification time ratio of our function. Then we run simulations showing how the number of cache misses during the execution of our memory-bound function depends on the cache size and the cache replacement strategy. We observe that even if an adversary knows future accesses, this does not help much unless the cache size is close to the size of $T$. Finally, we study how the running time depends on the size of the big array $T$.

## 7.1 Different Architectures

| name | class | model | processor | CPU clock | OS |
|---|---|---|---|---|---|
| P4-3060 | workstation | DELL XW8000 | Intel Pentium 4 | 3.06 Ghz | Linux |
| P4-2000 | desktop | Compaq Evo W6000 | Intel Pentium 4 | 2.0 Ghz | Windows XP |
| P3-1200 | laptop | DELL Latitude C610 | Intel Pentium 3M | 1.2 Ghz | Windows XP |
| P3-1000 | desktop | Compaq DeskPro EN | Intel Pentium 3 | 1.0 Mhz | Windows XP |
| Mac-1000 | desktop | Power Mac G4 | PowerPC G4 | 1000 Mhz | OSX |
| P3-933 | desktop | DELL Dimension 4100 | Intel Pentium 3 | 933 Mhz | Linux |
| SUN-900 | server | SUN Ultra 60 | UlraSPARC III+ | 900 Mhz | Solaris |
| SUN-450 | server | SUN Ultra 60 | UlraSPARC II | 450 Mhz | Solaris |
| P2-266 | laptop | Compaq Armada 7800 | Intel Pentium 2 | 266 Mhz | Windows 98 |
| S-233 | settop | GCT-AllWell STB3036N | Nat. Semi. Geode GX1 | 233 Mhz | Linux |

Table 1: Computational Platforms, sorted by CPU speed.

| machine | L2 cache | L2 line | memory |
|---:|---:|---:|---:|
| P4-3060 | 256 KB | 128 bytes | 4 GB |
| P4-2000 | 256 KB | 128 bytes | 512 MB |
| P3-1200 | 256 KB | 64 bytes | 512 MB |
| P3-1000 | 256 KB | 64 bytes | 512 MB |
| P3-933 | 256 KB | 64 bytes | 512 MB |
| Mac-1000 | 256 KB | 64 bytes | 512 MB |
| SUN-900 | 8 MB | 64 bytes | 8 Gb |
| SUN-450 | 8 MB | 64 bytes | 1 Gb |
| P2-266 | 512 KB | 32 bytes | 96 MB |
| S-233 | 16 KB | 16 bytes | 128 MB |

Table 2: Memory hierarchy.

We conducted tests on a variety of platforms, summarized in Table 1. These platforms vary from the popular Pentium 3 and Pentium 4 systems and a Macintosh G4 to SUN servers with large caches. We even tested our codes on a settop box, which is an example of a low-power device. The P2-266 laptop is an example of a "legacy" machine and is representative of a low-end machine among those widely used for e-mail today (that is, in 2003). Table 2 gives sizes of the relevant components of the memory hierarchy, including L2 cache size, L2 cache line size, and memory size. With one exception, all machines have two levels of cache and memory. The exception is the Macintosh, which has a 2 MB off-chip L3 cache in addition to the 256 KB on-chip L2 cache.

## 7.2 Memory- vs. CPU-Bound

| machine | HashCash | MBound | |
|---|---|---|---|
| name | time | time | sol./ver. |
| P4-3060 | 1.00 | 1.01 | 2.32 E4 |
| P4-2000 | 1.91 | 1.33 | 1.65 E4 |
| P3-1200 | 2.21 | 1.00 | 2.55 E4 |
| P3-1000 | 2.67 | 1.06 | 2.48 E4 |
| Mac-1000 | 1.86 | 1.96 | 2.61 E4 |
| P3-933 | 2.15 | 1.06 | 2.51 E4 |
| SUN-900 | 1.82 | 2.24 | 2.50 E4 |
| SUN-450 | 5.33 | 2.94 | 2.02 E4 |
| P2-266 | 10.17 | 2.67 | 1.84 E4 |
| S-233 | 43.20 | 4.62 | 1.50 E4 |

Table 3: Program timings. Times are averages over 20 runs, measured in units of the smallest average. For HashCash, the smallest average is 4.44 sec.; for MBound, it is 9.15 sec.%.

The motivation behind memory-bound functions is that their performance is less dependent on processor speed than is the case for CPU-bound functions. Our first set of experiments compares an implementation of our memory-bound function, *MBound*, to our implementation of *HashCash* [20]. HachCash repeatedly appends a trial number to the message and hashes the resulting string, until the output ends in a certain number zero bits (22 in our experiments). For MBound, with its slower iteration time, we set the required number of zero bits to 15.

Table 3 gives running times for HashCash and MBound, normalized by the fastest machine time. Note that HashCash times are closely correlated with processor speed. Running times for MBound show less variation. The difference between the P2-266 laptop and the fastest machine used in our tests for HashCash is a factor of 10.17, while the difference for MBound is only a factor of 2.67. The HashCash vs. MBound gap is even larger for the S-233 settop box.[15]

Modern Pentium-based machines perform well in memory-bound computations. The Macintosh does not do so well; we believe that this is due to its poor handling of the translation lookahead buffer (TLB) misses. SUN servers do poorly in spite of their large caches. This is due to their poor handling of TLB misses and the penalty for their ability of handle large memories.

## 7.3 Work Ratio

Recall that our experiments require 15 last bits of a hash of $A$ to be zero for the memory-bound puzzle to be solved. The expected number of paths we need to try is $2^{15}$. This is an upper

[15]Note that S-233 is a special-purpose device and code produced by the C compiler may be poorly optimized of the processor. This may be one of the reasons why this machine was so slow in our tests.

bound on the work ratio between the solver and the verifier. However, while for the solver initialization (dominated by computing a hash of a string that depends on the input message and customizing entries of $A$ for the input message) is well-amortized over the work involved in following the paths, for the verifier, who follows only one path, this work a non-trivial fraction of the running time.

The initialization overhead is a tradeoff between the size of $A$, the length of the input string, and the path length. In out experiments, $A$ contain 256 words, the path length is equal to 2048, and the input string contains 64 bytes. The last column of Table 3 gives the work ratio. Except for the settop box, the work ratio is greater than $2^{14}$, which means that the work involved in following the path dominates verification. For the settop box, the ratio is slightly less than $2^{14}$, meaning that initialization and path-following take approximately the same time.

These experimental results show that our parameter choices yield a reasonable work ratio.

## 7.4 Hit or Miss

The pattern of our accesses to $T$ is pseudorandom. In this section we present evidence that as long as the cache size is significantly smaller than the size of $T$, an adversary will gain little from knowing the access patters in advance. We show that if the adversary is constrained to follow the protocol, then even *optimal off-line* cache replacement, in which the cache line used farthest in the future is evicted from cache [7], does not significantly reduce the adversary's costs.

To do this, we simulate cache accesses for several cache sizes and cache replacement policies, including the optimal off-line one. We assume that $T$ takes 16 MB and that the cache size is at most 8 MB. Furthermore, in our simulations we assume that entries of $T$ are the only data in the cache.

If the access pattern to $T$ were truly random, then for any cache replacement policy and for any access, the probability of a cache hit is the ratio between the size of $T$ and the memory size. Our simulations confirm that this is the case for the LRU replacement policy. We believe that in time less than the memory latency, it is impossible to predict the next access address well, and therefore MBound will have the number of cache misses similar to that of a truly random access sequence.

However, since we cannot prove it is impossible to predict MBound's memory access pattern efficiently enough, we simulate the worst-case scenario where an adversary knows future and uses the optimal off-line cache replacement mechanism, "farthest in the future" [7]. For each cache line, this mechanism defines the next access time to be the *key*, and evicts the cache line with the largest key. We consider two types of cache, the fully associative (in which data from any address can be stored in any cache location) and the more realistic 8-way associative (in which the cache and the memory are partitioned into eight regions, and each address in memory may be stored in just one of eight different cache regions).

Table 4 gives cache miss ratios obtained by simulation. Even with optimal off-line scheduling, the 8-way cache gives limited improvement over LRU. If at most one quarter of $T$ fits in the cache, the miss ratio is above 50%. Even when half of $T$ fits, the miss ratio is relatively high. The optimal fully associative cache performs better. When half of $T$ fits in cache, the miss ratio, 18.8%, is lower than we would have liked. However, when a quarter of $T$ fits, the miss ratio grows to a more acceptable level.

| cache size | LRU | optimal | 8-way optimal |
|---|---|---|---|
| 8 MB | 50.0% | 18.8% | 30.7% |
| 4 MB | 75.0% | 38.4% | 51.3% |
| 2 MB | 87.5% | 54.5% | 65.7% |
| 1 MB | 93.8% | 66.9% | 75.8% |
| 512 KB | 96.9% | 76.2% | 82.9% |
| 256 KB | 98.4% | 83.0% | 87.9% |
| 128 KB | 99.2% | 87.8% | 91.5% |

Table 4: Cache miss ratios.

Note that even if our pseudorandom generation process is completely broken, in the sense that, even without accessing $T$, it is possible to completely predict the access sequence of a walk, it is unlikely that one can compute an optimal cache replacement strategy in time below the running time of MBound. Moreover, an approximately optimal schedule may lose most of the potential performance improvement.

The memory-bound approach assumes that there is a gap between the size of the memory of the least powerful machine and the size of the fast cache of the most powerful one. The results of this section suggest that this gap is enough for MBound to achieve its objective.

## 7.5 Size Matters

In this section we study MBound time as a function of the size of $T$ on a single machine. We use the P3-933 desktop for this experiment.

| $T$ kB | mem. access | MBound time |
|---|---|---|
| | sec | sec |
| 64 | 0.9 | 3.2 |
| 128 | 0.9 | 3.2 |
| 256 | 5.3 | 4.3 |
| 512 | 10.6 | 8.2 |
| 1024 | 10.6 | 10.5 |
| 2048 | 10.6 | 11.7 |
| 4096 | 10.6 | 12.4 |

Table 5: Dependence on the size of $T$.

Table 5 gives data for this experiment. Recall that the machine has 256 KB of L2 cache. When $T$ is smaller than the cache, MBound runs about four times faster than for $T$ of size 16 MB. In this case MBound is CPU bound, and the ratio depends on CPU performance. When $T$ is bigger than the cache size, MBound takes longer. The running times increase slightly as the size of $T$ increases because the cache miss ratio is growing, as discussed in the previous section. When the size of $T$ matches the L2 size, MBound performance is somewhat worse than for the smaller sizes, but not as bad as for larger ones. This is because, on one hand, L1 cache is present, and, on the other hand, $T$ competes for cache space with other variables and instructions of the program. The result is a moderate cache miss ratio.

Note that for the size of $T$ at and slightly above the cache size, the memory time access estimate is too high. This is because the estimate is obtained by using the worst-case memory access pattern and for the size of 512 MB and greater, all accesses to $T$ result in cache misses. When the size of $T$ matches L2 size, the presence of the L1 cache and the fact that other program data and instructions are accessed in a somewhat predictable way become critical. This explains why the performance in this case is between the 100% hit and the 100% miss cases.

## 8 Freeing the Receiver from Accessing $T$

Since the spam-protected receiver will sometimes act also as an e-mail sender, he will have access to the array $T$. However, we would like receiving mail not to have to involve accessing $T$ at all. For example, one might wish to be able to receive mail on a cell phone. In this section we explore the possibility that the sender adds some information to its message that will permit the receiver to efficiently verify the proof of effort with no accesses to $T$.

**Hash Trees.** A natural suggestion is to try *Merkle hash trees* [28] (see also [19, 32]). Consider a full binary tree, where the leaves correspond to the entries of $T$ and the value of each internal node is a an appropriately strong one-way hash of the value of its two children. Assume the receiver knows the value at the root. In order to prove the correctness of a value $T[c]$, the sender provides the values at the nodes on and adjacent to the path from the root to the leaf corresponding to $T[c]$[16]. This is repeated for every entry of $T$ that is to be verified (note that many of the root-leaf paths will have (relatively short) common prefixes).

How expensive is this scheme? First note that we can use universal one-way hash functions (UOWHFs) [30], since we only need to ensure that it is hard to find second preimages under the hash functions (this is sufficient because the entries in $T$ are not chosen maliciously; only the false ones are). For this kind of one-way hash function a smaller range (smaller than the one needed for collision-intractable hashing) suffices, resulting in a shorter proof. Assuming a range of $2^{80}$ elements and root-leaf path length $\log|T| = 22$, this means roughly $22 \cdot 80 = 1760$ bits per entry of $T$ in the successful path. Since the path length $\ell$ is taken to be roughly 1,000–2,000, this is an additional 200–400 KB per message. There are several ways to optimize this approach; for instance, we can assume that the receiver stores instead of just the root all nodes of depth, say 10, *i.e.*, roughly $1,000$ of them. This almost cuts in half the number of additional bits to be transmitted. Under this approach, verification requires $\ell \log|T|$ evaluations of a one-way hash function.

**Signature Schemes.** The *conceptually* simplest method for freeing $R$ from accessing $T$ is for the creator of $T$ to sign all the elements of $T$ (more precisely, the signature is on the pair $(c, T[c])$, to disallow permuting the table). However, this requires too much storage at the sender, even using the signature scheme yielding the shortest signatures [10].

[16]This is a path in the Merkle tree, not a path in the sense of our algorithm; to avoid confusion and emphasize the distinction we call these *root-leaf* paths.

**Compressed RSA Signatures.** Here we use properties of the RSA scheme previously exploited in the literature [15, 14]. Let $(N, e)$ be the public key of an RSA signature scheme chosen by the creator of $T$[17]. Let $F$ be a function mapping pairs $(c, T[c])$ into $Z_N^*$, that is, a mapping from $32 + 22 = 54$-bit strings into $Z_N^*$. In our analysis we will model $F$ as a random oracle. For all $1 \leq c \leq |T|$ let $v_c = F(c, T[c])$ and let $w_c = v_c^{1/e} \bmod N$. Thus, $v_c$ is a hash of the pair $(c, T[c])$ and $w_c$ is a signature on the string $v_c$.

The sender's protocol contains, in addition to $T$, the public modulus $N$, the description of $F$, and the $w_c$'s. The receiver's protocol uses only the description of $F$ and the public key $(N, e)$, together with a description of the sender's path exploration algorithm (minus the array $T$ itself).

Let the sender's successful path be the sequence $c_1, c_2, \ldots c_\ell$ of locations in $T$. The proof of effort contains two parts:

1. $T[c_1], T[c_2], \ldots, T[c_\ell]$, (a total of about 4 KB), and
2. $w = \prod_{i=1}^{\ell} w_{c_i} \bmod N$ (about 1 KB).

Note that there is no need to include the indices $c_1, \ldots, c_\ell$ in the first part, as these are implicit from the algorithm. Similarly, there is no need to send the $v_c$'s, since these are implicit from $F$ and the $(c_i, T[c_i])$ pairs. Let $t_1, \ldots, t_\ell$ be the first part of the proof, and $w$ the second part (each $t_i$ is *supposed* to be $T[c_i]$, but the verifier cannot yet be certain this is the case). The proof is checked as follows.

1. Compute $v'_{c_1}, v'_{c_2}, \ldots v'_{c_\ell}$, where $v'_{c_i} = F(c_i, t_i)$.
2. Check whether $w^e = (\prod_{i=1}^{\ell} v_{c_i}) \bmod N$.

The security of the scheme rests on the fact that it is possible to translate a forged signature on

$$(c_1, T[c_1]), \ldots (c_\ell, T[c_\ell])$$

into an inversion of the RSA function on a specific instance. This is summarized as follows:

**Theorem 2** *If $F$ is a random oracle, then any adversary attempting to produce a set of* claimed *values*

$$T[c_1], T[c_2], \ldots T[c_\ell]$$

*that is false yet acceptable to the receiver can be translated into an adversary for breaking RSA* with the same run time and probability of success *(to preserve probability of success we need that $e$ be a prime larger than $\ell$).*

Although transmission costs are low, the drawback of the compressed RSA scheme is again the additional storage requirements for the sender: each $w_c$ is at least $1,000$ bits (note, however, that these extra values are not needed until after a successful path has been found). This extra storage requirement might discourage a user from embracing the scheme. We address this next.

[17]The signing key $d$ is a valuable secret!

**Storage-Optimized Compressed RSA.** We optimize storage with the following storage / communication / computation tradeoff: Think of $T$ as an $a \times b$ matrix where $a \cdot b = |T|$; the amount of extra communication will be $a$ elements of $T$. The amount of extra storage required by the sender will be $b$ signatures.

At a high level, given a path using values $T[c_1], T[c_2], \ldots, T[c_\ell]$, values in the same row of $T$ will be verified together as in the compressed RSA scheme. The communication costs will therefore be at most $a$ elements, one per row of $T$. However, as we will see below, *there is no need to store the $w_c$'s explicitly.* Instead, we can get away with storing a relatively small number of signatures (one per column), from which it will be possible to efficiently reconstruct the $w_c$ values as needed.

Instead of a single exponent $e$, both sending and receiving programs will contain a (common) list of primes $e_1, e_2, \ldots e_a$. For $1 \leq i \leq a$, $e_i$ is used for verifying elements of row $i$ of the table. Although we don't need to store the $w_c$ values explicitly, for elements $v_c$ appearing in row $i$ we define $w_c = v_c^{1/e_i} \bmod N$.

The compressed RSA scheme is applied to the entries in each row independently. It only remains to describe how the needed $w_c$ values are constructed on the fly.

The $b$ "signatures", one per column, used in the sending program are computed by the creator as follows. For each column $1 \leq j \leq b$, the value for column $j$ is $u_j = \prod_{i=1}^{a} w_{c_{j_i}} \bmod N$. Here, $c_{j_i}$ is the index of the element $T[i,j]$, when $T$ is viewed as a matrix rather than as an array (that is, assuming row-major order, $c_{j_i} = (i-1)a + j - 1$). Thus, $v_{c_{j_i}} = T[(i-1)a + j - 1]$ and $w_{c_{j_i}} = (v_{c_{j_i}})^{1/e_i}$. As in Batch RSA [15], one can efficiently extract any $w_{c_{j_i}}$ from $u_j$ using a few multiplications and exponentiations.

Set $a = 16$. The number of data bits in a column is $2^4 \cdot 2^5 = 2^9$. The number of "signature bits" is $2^{10}$ per column. Thus storage requirement just more than doubles, rather than increasing by a factor of 5-10, at the cost of sending 16 elements of $Z_N^*$ (*i.e.*, 2 KB).

## 9 Concluding Remarks

We have continued the discussion, initiated in [8], of using memory-bound rather than CPU-bound pricing functions for computational spam fighting. We considered and analyzed several potential approaches. Using insights gained in the analyses, we proposed a different approach based on truly random, incompressible, functions, and obtained both a rigorous analysis and experimental results supporting our approach.

From a theoretical perspective, however, the work is not complete. First, we have the usual open question that arises whenever random oracles are employed: can a proof of security (in our case, a lower bound on the average number of cache misses in a path) be obtained without recourse to random oracles? Second, much more unusually, can we prove security without cryptographic assumptions? Note that we did not make cryptographic assumptions in our analysis.

One of the more interesting challenges suggested by this work is to apply results from complexity theory in order to be able to make rigorous statements about proposed schemes. One of the more promising directions in recent years is the work on lower bounds for branching program and the RAM model by Ajtai s [4, 5] and Beame et al [6]. It is not clear how to directly apply such results.

At first blush egalitarianism seems like a wonderful property in a pricing function. However, on reflection it may not be so desirable. Since the approach is an economic one it may be counterproductive to design functions that can be computed just as quickly on extremely cheap processors as on supercomputers – after all, we are trying to force the spammers to expend resources, and it is the volume of mail sent by the spammers that should make their lives intolerable while the total computational effort expended by ordinary senders remains benign. So perhaps less egalitarian is better, and users with weak or slow machines, including PDAs and cell phones, could subscribe to a service that does the necessary computation on their behalf. In any case, small-memory machines cannot be supported, since the large caches are so very large, so in any real implementation of computational spam fighting some kind of computation service must be made available.

**Acknowledgement.** We are grateful to Hoeteck Wee for helpful comments.

## References

[1] M. Abadi and M. Burrows, *(multiple) private communication(s)*

[2] M. Abadi, *private communication*

[3] InfoWorld http://www.infoworld.com/articles/hn/xml/02/08/13/020813hnbrin.xml

[4] M. Ajtai, *Determinism versus Non-Determinism for Linear Time RAMs*, STOC 1999, pp. 632–641.

[5] M. Ajtai, *A Non-linear Time Lower Bound for Boolean Branching Programs*, FOCS 1999: 60-70

[6] P. Beame, M. E. Saks, X. Sun, E. Vee, *Super-linear time-space tradeoff lower bounds for randomized computation,* FOCS 2000, pp. 169–179.

[7] L. A. Belady, *A Study of Replacement Algorithms for Virtual-Storage Computer* IBM Systems Journal 5(2), pp. 78–101, 1966

[8] M. Abadi, M. Burrows, M. Manasse, and T. Wobber, *Moderately Hard*, Memory-Bound Functions, Proceedings of the 10th Annual Network and Distributed System Security Symposium February, 2003.

[9] N. Alon, and J. Spencer, *The Probabilistic Method*, Wiley & Sons, New-York, 1992

[10] D. Boneh, B. Lynn and H. Shacham, *Short signatures from the Weil pairing*, ASIACRYPT 2001, pp. 514-532.

[11] www.camram.org/mhonarc/spam/msg00166.html.

[12] W. Diffie and M.E. Hellman, Exhaustive cryptanalysis of the NBS Data Encryption Standard, Computer 10 (1977), 74-84.

[13] C. Dwork and M. Naor, *Pricing via Processing, Or, Combatting Junk Mail*, Advances in Cryptology – CRYPTO'92, Lecture Notes in Computer Science No. 740, Springer, 1993, pp. 139–147.

[14] C. Dwork and M. Naor, *An Efficient Existentially Unforgeable Signature Scheme and Its Applications*, Journal of Cryptology 11(3): 187-208 (1998)

[15] A. Fiat, *Batch RSA*, . Journal of Cryptology 10(2): 75-88 (1997).

[16] A. Fiat and M. Naor, Rigorous Time/Space Tradeoffs for Inverting Functions, STOC'91, pp. 534–541

[17] A. Fiat and A. Shamir, How to Prove Yourself, *Advances in Cryptoglogy – Proceedings of CRYPTO'84*, pp. 641–654.

[18] Fluhrer, Mantin and Shamir, *Attacks on RC4 and WEP*, Cryptobytes 2002.

[19] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, **Handbook of Applied Cryptography**, CRC Press, 1996. Also available: `http://www.cacr.math.uwaterloo.ca/hac/`

[20] A. Back, Hashcash - A Denial of Servic Counter-Measure, available at `http://www.cypherspace.org/hashcash/hashcash.pdf`.

[21] M. Bellare, J. A. Garay and T. Rabin, *Fast Batch Verification for Modular Exponentiation and Digital Signatures*, EUROCRYPT 1998, pp. 236–250.

[22] M. Bellare, J. A. Garay and T. Rabin, *Batch Verification with Applications to Cryptography and Checking*, LATIN 1998, pp. 170–191.

[23] M. Hellman, A Cryptanalytic Time Memory Trade-Off, *IEEE Trans. Infor. Theory 26*, pp. 401-406, 1980

[24] A. Joux and J. Stern, Lattice Reduction: A Toolbox for the Cryptanalyst, *J. Cryptology 11*(3), pp. 161–185, 1998

[25] J. Lagarias, A. Odlyzko, Solving Low-Density Subset Sum Problems, *JACM 32*(1), pp. 229–246, 1985

[26] M. Luby and C. Rackoff, How to Construct Pseudorandom Permutations and Pseudorandom Functions, *SIAM J. Computing 17*(2), pp. 373–386, 1988

[27] I. Mantin, *Analysis of the Stream Cipher RC4*, Masters Thesis, Weizmann Institute of Science, 2001. Available `www.wisdom.weizmann.ac.il/~itsik/RC4/rc4.html`

[28] R. Merkle, *Protocols for Public Key Cryptography*, IEEE Symposioum in Security and Privacy, 1980, pp. 122–134.

[29] I. Mironov, (Not So) Random Shuffles of RC4, *Proc. of CRYPTO'02*, 2002

[30] M. Naor and M. Yung, *Universal One-Way Hash Functions and their Cryptographic Applications*, STOC 1989, pp. 33–43

[31] P. Oechslin, *Making a faster Cryptanalytic Time-Memory Trade-Off, CRYPTO 2003.*

[32] P. C. van Oorschot and M. J. Wiener, *Parallel Collision Search with Cryptanalytic Applications*, Journal of Cryptology, vol. 12, no. 1, 1999, pp. 1-28.

[33] R. Schroeppel and A. Shamir, *A* $T = O(2^{(n/2)})$, $S = O(2^{(n/4)})$ *Algorithm for Certain NP-Complete Problems*, SIAM J. Comput. 10(3): 456-464 (1981). 1979.