On Memory-Bound Functions for Fighting Spam

Cynthia Dwork[1], Andrew Goldberg[1], and Moni Naor[2]⋆

[1] Microsoft Research, SVC
1065 L'Avenida
Mountain View, CA 94043
{dwork,goldberg}@microsoft.com
[2] Weizmann Institute of Science
Rehovot 76100, Israel
naor@wisdom.weizmann.ac.il

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 speci£c 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 £rst explored by Abadi, Burrows, Manasse, and Wobber [3].
We further investigate this intriguing proposal. Speci£cally, 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; speci£cally, 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 signi£cantly 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.

⋆ Incumbent of the Judith Kleeman Professorial Chair. Research supported in part by a grant from the Israel Science Foundation. Part of this work was done while visiting Microsoft Research, SVC.

This paper focuses on the computational approach to £ghting spam, and, more generally, to combating denial of service attacks, initiated by Dwork and Naor [11] (also discussed by Back; see [18, 9]). 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 ¤avor; 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.

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, £rst explored in [3], 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 [21, 14] (where space now refers to cache) constrain the functions proposed in [3] (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)[3]. We suggest a very ef£cient 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 (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. A more complete version of the paper is available at
`www.wisdom.weizmann.ac.il/~naor/PAPERS/dgn.html`.

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

[3] None of [11, 18, 9, 3] obtains a lower bound.

hard to compute "pricing" function f. The message m is transmitted together with the other arguments to f and the resulting proof of effort z[4]. 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-prespeci£ed action, such as placing the message in a special folder, marking it as spam, subjecting it to further £ltering, and so on. Proof computation and veri£cation 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 £ghting 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 veri£cation very cheap, ensuring that the ability to wage a denial of service attack against a receiver is not exacerbated by the spam-£ghting tool. In addition, if veri£cation is suf£ciently 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: once the mail is sent the sender never need take any further action, once the mail is received the proof of effort can be checked locally; neither sender nor receiver ever need contact a third party. In other words, single-pass send-and-forget means that e-mail, the killer application of the Internet, is minimally disturbed.

Remark 2. We brie¤y 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 £rst checks the date: if the date is, say, over a week old, then the proof is rejected. Otherwise, the 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 [11], f is a forged signature in a careful weakening of the Fiat-Shamir signature scheme. Back's proposal, called *HashCash*, is based on £nding hash collisions. It is currently used to control access to bulletin boards [18]; veri£cation is particularly simple in this scheme.

3 Computational Model and Statement of the Problem

The focus on memory-bound functions requires speci£cation 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

[4] Having m as an argument to the function introduces some practical dif£culties 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.

majority) and *cache* – a small amount of storage on the same integrated circuit chip as the central processing unit[5]. 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.

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,

[5] In fact, there are multiple levels of cache; Level 1 is on the chip.

fast, on-chip cache. However, since the computational approach to £ghting 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 the full paper we show that pricing functions based on meet in the middle and subset sum can be computed with very few memory access, and hence do not solve our problem. In these proceedings we con£ne our attention to the proposal in [3], described next.

Easy-to-Compute Functions. These functions 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 ℓ be parameters and let $g : \{0,1\}^n \longrightarrow \{0,1\}^n$. Let g_0 be the identity function and for $i = 1 \dots \ell$, 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* α*, a hash of the values* $x, g_1(x), \dots, g_\ell(x)$.
Output. $x' \in g_\ell^{-1}(y)$ *such that the string* $x', g_1(x'), \dots, g_\ell(x')$ *hashes to* α.
The hope is that the best way to resolve the challenge is to build a table for g^{-1} and to work 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 limitation of this approach is that, since g can be computed with no memory accesses, there is a time/space tradeoff for *inverting* g in which no memory accesses are performed (in our context, space refers to cache size, since we are interested in what can be done without going to memory) [21, 14, 26]. Those results imply that g can be inverted at the cost of two forward computations of g, *with no memory accesses*.

This suggests basing computational challenges on functions that are (in some sense) *hard in both directions*.

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 de£ned 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,

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

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 [22] did for the application of Feistel Permutations in the design of DES; (ii) If there is an attack on the simpli£ed instantiation of the algorithm of Section 6, then the model provides guidelines for modi£cations. 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 [21, 14] do not apply to truly random functions, as these have large Kolmogorov complexity (no short representation). Accordingly, our function involves a large *£xed forever* table T of truly random w-bit integers[7]. 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 identi£ed, information enabling the receiver to check that a successful path has been found is sent along with the message. Veri£cation requires work proportional to the path length, determined by a parameter ℓ. Each path exploration is called a *trial*. The expected number of trials to £nd 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 modi£able 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)[8].

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, 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:

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

[8] The intuition for requiring $|A|w > b$ is that, since A cannot £t 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.

```
 Initialization:
  A = H_0(m, R, S, d, k)
Main Loop: Walk for ℓ steps (ℓ is the path length):
  c ← H_1(A)
  A ← H_2(A, T[c])
Success occurs if:
  after ℓ 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 £ve parameters and the £nal $H_3(A)$ obtained during the successful trial[9].

Veri£cation that the path is indeed successful is trivial: the veri£er 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: we 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 $|A|$ suf£ciently 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 £nd a successful path. Speci£cally, 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 speci£c 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 speci£ed calls. *By watching an execution unfold, we can observe when paths begin, and when they make progress.* Calls to the oracle that make progress (as determined by the history) are called *progress calls*.

Theorem 1. *Consider an arbitrarily long but £nite execution of the adversary's program – we don't know what the program is, only that the adversary is constrained to use an architecture as described in Section 3. Under the following additional conditions, the amortized complexity of generating a proof of effort that will be accepted by a veri£er is $\Omega(2^e \ell)$:*

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

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

Proof. (of Theorem 1) We start with an easy lemma regarding the number of oracle calls needed to £nd 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 veri£er is $\Omega(2^e\ell)$.*

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 $\{b_i \mid i \in S\}$. 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 ¤ips of the hint generation and the guessing system.*

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.

We £rst motivate our de£nition 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 modi£ed at each step of the Main Loop; intuitively, since these modi£cations require many elements of T, these "mature" A's cannot be compressed. Our de£nition 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 £t into cache (assuming they are incompressible). 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). An *interval* is de£ned by £xing 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.*

It is an easy consequence of this lemma that the amortized number of memory accesses to £nd 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 de£nition 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 β 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 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 βb 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 £rst 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. The proof is based on Sauer's Lemma (see [6])

Claim 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.

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 con£guration to show that it is possible to extract many entries of T'.

Claim 2. If the number β 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 $3n$.

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 speci£ed by the £rst 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 £rst 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 £nd an expected $(|T| - s)/|T| > 1/2$ of them.

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, for some £xed $\epsilon > 0$, the spammer completes the interval retrieving at most 2β blocks with at least ϵ probability, over the choice of $8n$-tuple. Call these tuples the *good* ones. By Markov's inequality, for at least half of these good $8n$-tuples the spammer retrieves at least β blocks. We £rst claim that in most of those tuples the spammer goes frequently into H_2 with values $T[i]$ where $i \in T'$.

Claim 3. Let T' be any subset of the entries of T of size at least δn. Consider the set of good $8n$-tuples over $\{1, \ldots, |T|\}$ as the set of possible answers H_1. Then except for at most an exponential in n fraction of them the spammer must use an entry in T' for a call to at H_2 least n times during an interval.

Claim 4. Suppose that we have subset X of size x of entries in T. Then the probability over H_1 that a $8n$-tuple contains more than $n/2$ entries in X is at most $(2^8 x/|T|)^{n/2}$.

Claim 5. Suppose that we have a collection of good $8n$-tuples and we want to cover at least x values in T' using only a few members of the collection, say $2x/n$ (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.*

The idea for deriving the contradiction to the fact that only $\beta = o(n)$ blocks are brought from memory to cache is that there should 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). In fact, since the memory size is m, a $1/\binom{m}{2\beta}$ fraction of them share the same set of blocks (the factor of 2 comes from the de£nition of a good $8n$-tuple). Consider such a collection and suppose that there are $2x/n$ tuples in this collection whose union covers x entries in T'. Then the "memory" can use these $2x/n$ tuples to transfer the value of x entries in T' by sending the $2\beta b$ bits describing the content of the common blocks and in addition for each tuple in the cover: (1) Specifying the $8n$-tuple: this takes $8n \log |T|$ bits; (2) 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 takes $\log \binom{z}{n/2}$ bits which is $O(n \log z)$.

So altogether it su£ces for $2\beta b + 16x \log |T| + 2x \log z$ bits to be sent from the memory to the cache. In return, the cache learns γw bits for each of x entries in T', or $x\gamma w$ bits altogether. To derive the contradiction, 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 $\beta \le 1/20n = s/20|A|$ and, for simplicity that m, the memory size is $|T|^2$ (recall that in our model m is polynomial in s, and in our theorem $|T| = \Theta(s)$). Set $x = 4\beta b/w$. Of all good tuples, pick the largest collection agreeing with a set of β blocks, *i.e.*, consisting of at least a $1/\binom{T^2}{\beta}$ fraction of the good tuples. We now claim that this collection has $2x/n$ $8n$-tuples whose union is of size at least x (this will be su£cient for a contradiction).

Suppose that this is not the case and the $2x/n$ tuples covering x do not exist. Then as we have seen above in Claim 5 there is a set X of size x where each tuple in the collection has at least $n/2$ entries in X. But we know from Claim 4 that the fraction (among all tuples) of such a collection can be at most $(2^8 x/|T|)^{n/2}$. Taking into account ϵ (the faction of all tuples that are good) we must compare $((2^8 x/|T|)^{n/2})(1/\epsilon)$ to $1/\binom{T^2}{2\beta}$ and if the latter is larger we know that the collection is too large to be compressed into X. For simplicity take $\epsilon = 1$. Indeed

$$\frac{(2^8 x/|T|)^{n/2}}{\binom{T^2}{2\beta}} = \frac{(2^8 x)^{n/2}}{T^{n/2-4\beta}}$$

taking logs we get that we need to compare $\log 2^8 x$ and

$$(\log T)\frac{n-4\beta}{n} = (\log T)\frac{s/|A| - 4s/20|A|}{s/|A|} = (\log T)\frac{4}{5}.$$

But since $x = 4\beta b/w = 4sb/(20|A|w)$ and $|A| \ge s^{1/5} b/w$ we get that $x \le 1/5 s^{4/5}$ and indeed $8 + \log x$ is smaller than $4/5 \log |T|$.

This concludes the proofs of Lemma 3 and Theorem 1

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 *£xed forever* array T, now of 2^{22} truly random 32-bit integers[10]. 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.[11] The algorithm requires in addition a £xed-forever truly

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

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

random array A_0 containing 256 32-bit words. A_0 is used in the de£nition 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 [16, 23, 24]).

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

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 α_k with itself 2^6 times[12]. 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 $\{1, 2, \dots, 256\}$, 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 ℓ steps (ℓ 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 modi£cation of A and in the updating of c.

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

[12] 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_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).

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

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 de£ne 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 α such that $1 \leq i \leq 2e$ and the ith path (that is, the path with trial number $k = i$), is successful and α is the result of hashing the £nal value of A in the successful path.

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

To be speci£c in the following analysis, we make several assumptions. These assumptions are reasonable for current technology, and our analysis is suf£ciently 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 τ be the memory latency. We assume that P is 10 seconds and τ 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 veri£cation by the receiver is essentially ℓ 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 ℓ. 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 veri£cation 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 modi£ed. 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 [18] on a variety of computer architectures. We con£rm that the memory-bound function performance is signi£cantly more platform-independent. We also measure the solution-to-veri£cation 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.

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.

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.

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 £rst set of experiments compares an implementation of our memory-bound function, *MBound*, to our implementation of *HashCash* [18]. 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.[13]

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.

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 ef£ciently verify the proof of effort with no accesses to T. Of course, 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 [8].

Compressed RSA Signatures. Here we use properties of the RSA scheme previously exploited in the literature [13, 12]. Let (N, e) be the public key of an RSA signature scheme chosen by the creator of T[14]. 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 $c_1, c_2, \dots c_\ell$ of locations in T. The proof of effort contains two parts:

1. $T[c_1], T[c_2], \dots, T[c_\ell]$, (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 $c_1, \dots, c_\ell$ in the £rst 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 $t_1, \dots, t_\ell$ be the £rst part of the proof, and

[13] 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.

[14] The signing key d is a valuable secret!

w the second part (each t_i is *supposed* to be $T[c_i]$, but the veri£er 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 speci£c 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 ℓ).*

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.

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 veri£ed 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 ef£ciently 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 de£ne $w_c = v_c^{1/e_i} \bmod N$.

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 ¤y.

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 [13], one can ef£ciently 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 [3], of using memory-bound rather than CPU-bound pricing functions for computational spam £ghting. 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 [7]. It is not clear how to directly apply such results.

At £rst blush egalitarianism seems like a wonderful property in a pricing function. However, on re¤ection 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 £ghting some kind of computation service must be made available.

References

1. M. Abadi and M. Burrows, *(multiple) private communication(s)*
2. M. Abadi, *private communication.*
3. 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.

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. N. Alon, and J. Spencer, *The Probabilistic Method*, Wiley & Sons, New-York, 1992
7. P. Beame, M. E. Saks, X. Sun, E. Vee, *Super-linear time-space tradeoff lower bounds for randomized computation,* FOCS 2000, pp. 169–179.
8. D. Boneh, B. Lynn and H. Shacham, *Short signatures from the Weil pairing*, ASIACRYPT 2001, pp. 514-532.
9. www.camram.org/mhonarc/spam/msg00166.html.
10. W. Dif£e and M.E. Hellman, Exhaustive cryptanalysis of the NBS Data Encryption Standard, Computer 10 (1977), 74-84.
11. 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.
12. C. Dwork and M. Naor, *An Ef£cient Existentially Unforgeable Signature Scheme and Its Applications*, Journal of Cryptology 11(3): 187-208 (1998)
13. A. Fiat, *Batch RSA*, . Journal of Cryptology 10(2): 75-88 (1997).
14. A. Fiat and M. Naor, Rigorous Time/Space Tradeoffs for Inverting Functions, STOC'91, pp. 534–541
15. A. Fiat and A. Shamir, How to Prove Yourself, *Advances in Cryptoglogy – Proceedings of CRYPTO'84*, pp. 641–654.
16. Fluhrer, I. Mantin and Shamir, *Attacks on RC4 and WEP*,. Cryptobytes 2002.
17. 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/`
18. A. Back, Hashcash - A Denial of Servic Counter-Measure, available at `http://www.cypherspace.org/hashcash/hashcash.pdf`.
19. M. Bellare, J. A. Garay and T. Rabin, *Fast Batch Veri£cation for Modular Exponentiation and Digital Signatures*, EUROCRYPT 1998, pp. 236–250.
20. M. Bellare, J. A. Garay and T. Rabin, *Batch Veri£cation with Applications to Cryptography and Checking*, LATIN 1998, pp. 170–191.
21. M. Hellman, A Cryptanalytic Time Memory Trade-Off, *IEEE Trans. Infor. Theory 26*, pp. 401-406, 1980
22. M. Luby and C. Rackoff, How to Construct Pseudorandom Permutations and Pseudorandom Functions, *SIAM J. Computing 17*(2), pp. 373–386, 1988
23. 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`
24. I. Mironov, (Not So) Random Shuf¤es of RC4, *Proc. of CRYPTO'02*, 2002
25. M. Naor and M. Yung, *Universal One-Way Hash Functions and their Cryptographic Applications*, STOC 1989, pp. 33–43
26. P. Oechslin, *Making a faster Cryptanalytic Time-Memory Trade-Off*, these proceedings.
27. 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.
28. 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.