Exponential Memory-Bound Functions for Proof of Work Protocols

Technical Report A/370/CRI version 2

Fabien Coelho
CRI, École des mines de Paris,
35, rue Saint-Honoré, 77305 Fontainebleau, France.
fabien.coelho@ensmp.fr

Abstract

In year 2005, Internet users were twice more likely to receive unsolicited electronic messages, known as spams, than regular emails. *Proof of work* protocols are designed to limit such phenomena and other denial-of-service attacks by requiring some kind of virtual stamping: the attacks are not suppressed, but their rate is reduced as some cost must be paid to get the service. The cost is not in money but in computation time. These schemes require computing an easy to verify but hard to find solution to some problem. As cpu-intensive computations are badly hit over time by Moore's law, memory-bound computations have been suggested as an alternative to deal with heterogeneous hardware. We introduce new memory-bound functions suitable to these protocols, in which the client-side work to compute the response is exponential with respect to the server-side work needed to set the challenge or check it, instead of polynomial. One-way non-interactive solution-verification variants are also presented. Our experimental results and technical arguments show that any such memory-bound function is inherently parallel, thus bound by memory bandwidth and not by memory latency, as previously claimed by others.

1 Introduction

In recent years, the Internet electronic mail user has been plagued with massively sent unsolicited messages, *a.k.a.* spams [16]. This communication channel has proven interesting to marketers thanks to its combined worldwide-spread use in upscale households and to the very low sender-side cost of messages: two thirds of all emails were spams [15] in 2005. Many approaches [11] have been investigated to tackle this phenomenon. They aim at preventing, deterring, detecting or responding to it appropriately. Automatic contents and behavior filtering [14] is in place in most organizations to reduce the burden of handling these messages for users and systems.

We focus here on a particular cryptographic technique by Dwork and Naor [4] which suggests to put a tighter economic bound to spamming by making emails more expensive to send, thanks to some kind of stamps. Spams are not suppressed, but are simply made more costly. The suggested stamps are not actual money, but rely on a proof of computation work performed by the sender, on top of the popular saying that *time is money* [5]. Hashcash [2] is such a scheme. The client builds a stamp for a service, such as sending a mail to an address today, by producing a bit-string. The hashed value of this bit-string added to the service description must start with a number of leading zeros, depending on the expected work. The high solution cost comes from enumerating many bit-strings up to one having the required partial hash collision property, although the verification is one straightforward computation on the provided solution bit-string.

Economical measures to contain denial-of-service attacks have been pursued for other purposes than deterring spams: proof of work allows to introduce delays [18]; it is a tool to audit the reported metering of web-sites [6]; puzzle resolutions [12] or auctions [21] are used to limit the incoming flow of service requests; in [19], a protocol for preserving digital data relies on such schemes to resist malign peers; a formalization of proof of work schemes is presented by [10]; finally, actual financial analysis are suggested [13] as useful to evaluate the impact of these techniques on a particular problem.

Proof of work scheme variants may include interactive challenge-response protocols shown in Figure 1, or one-way solution search followed by a verification in Figure 2.

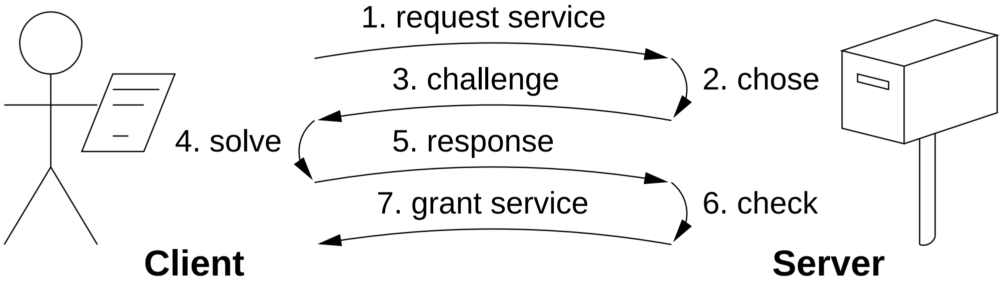

Figure 1: Interactive Challenge-Response Protocol

The challenge-response approach better suits synchronous end-to-end client-server protocols, where a server regulates its own incoming traffic. As there is no direct end-to-end connections in the electronic mail realm where intermediate Mail Transfer Agents (MTA) handle messages, it could be used at the entry point of a trusted network of such agents built by other means. The client-server sides are somehow reversed in the inner protocol, as the server which is requested the final service does request the proof of work, and the client which want to be granted the service must respond to the work request.

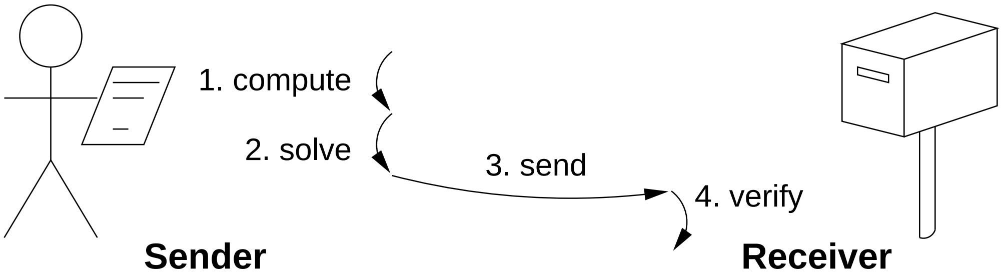

Figure 2: One-way Solution-Verification Protocol

The solution-verification approach is well fitted to asynchronous off-line message composition and later transfer, delivery or user filtering, where the stamp may be an element among others to filter out some messages depending on the policy of the agent. It does not need interaction between the sender requesting the service and the final service provider. However, as the problem to solve is self-imposed by the sender, it must depend somehow on the requested service, and the receiver must both check the validity of the chosen problem and of the provided solution.

A key issue of both approaches is to compare the amount of work required of the client to compute the stamp in the response or solution part with respect to the work expected of the server to set the challenge or verify the provided solution. Another issue is that although interactive challenge-response protocols can lead to a known-bound search effort, as the server chooses an existing target in the search space, one-way solutions are usually bound in the probabilistic sense. For instance in hashcash-like methods a user will statistically spend more than 4 times the average time for computing one solution every $e^4 \approx 55$ mails: one's laptop may suddenly hangs for one minute every day when sending an email.

Processor computational performance varies more widely than cache to memory access performance [22, 7] from high-end servers to low-end PDAs (personal digital assistant) and over time, following Moore's law [17]. Thus, Abadi *et al.* [1] suggests to implement a scheme based on memory-bound functions, the performance of which are bound by main memory access speed instead of cpu and cache accesses. This approach is further investigated by Dwork *et al.* [3].

This paper presents new contributions about memory-bound proof of work functions. It is organized as follows: Section 2 presents and analyzes related work by Abadi *et al.* and Dwork *et al.* Section 3 describes our new challenge-response Hokkaido protocol for memory-bound proof of work schemes. For a challenge cost of $\mathcal{O}(l)$ the response cost is $\mathcal{O}(2^l)$ memory accesses, thus inducing an exponential work for the client compared to the server. This exponential memory-bound behavior is obtained by using a mangled path in a binary tree on a tabulated function. Section 4 details one-way variants for the same purpose. It also emphasizes new choices for setting the various message-dependent parameters when used in an anti-spam context. Section 5 contributes experimental results and discusses technical issues. It shows that these algorithms are necessarily bound by memory bandwidth, but not memory latency, as previously claimed. Code optimization issues related to the implementation of these functions are addressed. Finally, Section 6 concludes this paper.

2 Related work

Abadi *et al.* [1] describe a challenge-response protocol in which the server about to receive an email requires the client wanting to send it to perform $\mathcal{O}(l^2)$ memory accesses, although the verification costs $\mathcal{O}(l)$ computations, where l is a length parameter. The challenge setting phase computes a sequence $x_{i,0\leq i\leq l}$ starting from a chosen x_0:

$$x_{i+1} = f(x_i) \otimes i$$

where f is a random-like function on a domain and $\otimes$ the exclusive-or operator. The response is to look for x_0, by computing the reverse path starting backwards from x_l:

$$x_{i-1} \in f^{-1}(x_i \otimes (i-1))$$

and checking the path to x_0 against a provided checksum. Note that f^{-1} is not a simple function: there may be several pre-images at each stage leading to multiple paths… indeed, the response to challenge work cost ratio is achieved by forcing the client to explore this tree of reverse paths using the tabulated inverse of the function, while the verification uses simpler forward computations. If the function domain is large enough, *e.g.* 2^{22} elements, the tabulated inverse does not fit into the cache, and many costly main memory accesses are performed.

The proposed technique is original, but has several drawbacks. First, the solution cost is *only* quadratic, thus sizable amount of verification cost is required by the server if the client is to provide a proof of significant work. A large value $l = 2^{13} = 8192$ is suggested. Second, this quadratic behavior actually depends on the chosen function to *be* random *and* on the forward path to be *known* to exists: for a permutation, the response work is the same as the verification work because only one reverse path exists; for a random function without known forward path, only one reverse path exists on average, hence there is no quadratic effect. Third, the actual multiplier hidden by the $\mathcal{O}()$ notation is $\frac{1}{2(e-1)} \approx 0.3$ for purely random functions. This is small: there are few reverse paths because the average number of pre-image by a function on a domain is just one. Fourth, the data structures needed for handling the inverse of random functions on an arbitrary domain are not really nice: either fast lookups can be used but memory is wasted, or a packed representation is used at a higher computation cost... as the aim is to maximize memory random accesses compared to computations, this is annoying.

Dwork *et al.* [3] propose a non-interactive one-way scheme inspired by the RC4 cipher. The verification cost is $\mathcal{O}(l)$ for an exploration work of $\mathcal{O}(E.l)$ memory accesses. Although a simple constant work ratio seems less interesting than the previous proposal, E can be chosen as a nearly-arbitrary large *effort* parameter.

The scheme performs solution-seeking trials based on an integer parameter k till a solution satisfying some property is reached. For trial number k, an initial state s_0 is computed from the result of a cryptographic-strong hash function h applied on the message or service m and k:

$$s_0 = \text{init}(h(m, k))$$

Then the state is updated l times with a function that performs one lookup into a large constant random integer table t:

$$s_{i+1} = \text{update}(s_i, t(r(s_i)))$$

The final state s_l is a success if some $\frac{1}{E}$-probable property holds for $h(s_l)$. Verifying a solution requires to perform the full trial computation again for the provided parameter k.

The authors suggest $l = 2^{11} = 2048$: such a large value is needed to amortize the initialization phase and the cryptographic-strong hash computations that occur at each trial. It is claimed that this value allows the function to be memory-latency bound, but we show that it is not. An effort parameter $E = 2^{15} = 32768$ is proposed to achieve a significant work for the solution seeking process, so that about 2^{26} table accesses occur.

The next three sections present our contributions. We first introduce new challenge-response memory-bound functions similar to Abadi *et al.*, but with much better exponential client-to-server work ratio. Then we describe one-way solution-verification variants with faster checking costs compared to Dwork *et al.* proposal. Finally, experiments and analyses illustrate our achievements on practical examples.

3 The Hokkaido protocol

Our Hokkaido[1] protocol is first introduced, then various choices of parameters and their implications are discussed.

3.1 Challenge-response protocol

Let D be a finite integer domain. Its size $n = |D|$ is typically a power of two. Let f be a function from D to D. The next subsection discusses what this function might be. Let l be the path length, which will be quite short in practice. The server chooses l non-zero elements in D $n_{i,1 \leq i \leq l}$, l booleans b_i and a starting point x_0 also in D. It computes l iterations on f:

$$x_{i+1} = f(x_i) \otimes (\text{if } b_i \text{ then } n_i \text{ else } 0)$$

Elements n_i must not be zero so that x_{i+1} always differ depending on b_i. Then the challenge is composed of f, l, $n_{i,1 \leq i \leq l}$, the end point x_l and a checksum c of the path from x_0. The response is the starting point x_0 and the binary path b_i which leads to x_l and matches the path checksum.

This looks much like the Abadi *et al.* function, as the server performs a straight forward computation to set its challenge and the client is required to find a reverse path in a tree for the response. The actual computations performed are minimal. However, unlike the previously proposed scheme, the reverse tree is built so that its size is intrinsically exponential, without additional assumptions such as the randomness of Function f for a quadratic only result. The key idea is that thanks to the random-path mangling of values at each stage, every elements has two different images through f, and thus will have on average two pre-images through f^{-1} when going backwards.

A possible algorithm to compute the response in $\mathcal{O}(n.2^l)$ is to try all forward paths from all starting points. A more interesting algorithm is to perform a backward search from x_l using f^{-1}. Due to the possible scrambling by n_i at each stage, each point has 2 predecessors on average, hence the complexity is at least in $\mathcal{O}(2^l)$. For a constant function, the complexity is $\mathcal{O}((2n)^l)$. As argued by Abadi *et al.*, if a large tabulated representation of f^{-1} is necessary, then many slow memory accesses occur, hence the computation is memory bound.

3.2 Discussion

Let us now discuss various choices of parameters in the above algorithm. Possible choices for Function f and Domain D are outlined. Implementation issues are addressed, with respect to the data structures needed for f^{-1}. Checksum variants are discussed, esp. compared to their cost in the search.

f as a random-like function

Function f might be a computed random-like function as suggested by Abadi *et al.*, the issue being to devise a very fast function to compute forward without any computable inverse apart from memory-hungry tabulated values. If bits are not stewed a lot in the forward computation, it might be feared that some computation may be found to shortcut the inverse. If bits are really mangled

[1] A nice place in Japan to think about memory-bound functions.

as in cryptographic-strong hash functions, then the cost of the forward computations to build the inverse table might not be negligible at all. Some trade-off must be made.

When tabulating the inverse of an arbitrary function on a domain, care must be taken to deal with collisions. Using a standard hash table data structure is not interesting because it requires a large memory overhead for pointers. An astute data structure for the inverse of any function in a domain of size n can be built in $\mathcal{O}(n)$ using at most $\lceil \log_2(n) \rceil (2n+1)+1$ bits of storage with fast $\mathcal{O}(1)$ enumeration costs, assuming that values of size $\lceil \log_2(n) \rceil$ in Domain D can be handled directly in $\mathcal{O}(1)$. First, a boolean tells whether the function is a permutation. If so, a first bit-aligned array of size $\lceil \log_2(n) \rceil n$ simply holds the tabulated inverse permutation. If not, at least one collision occurs in f, so at least one value in the inverse has several pre-image. One such pre-image is used as a special value in the first bit-aligned array to tag elements without pre-image. For those elements with one pre-image, a second similar bit-aligned array gives other indexed pre-images through an array-encoded linked list. Thus the factor 2 is needed to tabulate if each element has an inverse, and then when an inverse is found, whether others are available.

f as a permutation

If f is a permutation, then it cannot be a fast computed permutation, as its inverse would certainly be easy to compute. So let us assume that f is a *tabulated* permutation. A possible algorithm to build such a permutation is to initialize f's table to the identity, and then to exchange every element in turn with another pseudo-randomly chosen one, generating n pseudo-random table accesses on the way of the building process. The memory needed to store compactly permutation f with indexed accesses is about $\lceil \log_2(n) \rceil n$ bits, half the amount required for an arbitrary function.

Now, if f is already a tabulated permutation, it is useless to compute its tabulated inverse, also a permutation: a simpler forward variant of our Hokkaido protocol can be devised where x_0 is given instead of x_l in the challenge, and the response must find the end point and the path leading to it which matches the checksum. An advantage of this approach is that it is much less likely that some hidden computable shortcut in a pseudo-randomly built tabulated permutation exists. Some drawbacks are that the server must compute the table in order to set and verify a challenge: the table may be constant on the server so that it can be reused from one challenge to the other and thus its computation amortized. On the other hand, the building process from some provided seed may be part of the proof of work of the response, as it also involves random memory accesses. For the forward computation part of setting a challenge, it must be noted that the expected length will be quite small due to the exponential work of response, hence the cost of cache misses is expected to be very small on the server side once the table is available in the main memory.

Implementation issues

The aim of memory bound functions is to be memory bound. This is not as trivial as it seems when one addresses implementation issues. Indeed, the time to run a search is shared both by computations and accesses to memory needed for the computations. Although memory accesses are slow, they are not tremendously slow. Thus as few computation as possible must be required if the computation time is to be really small in front of memory accesses. The processor can easily linger in other tasks on the critical path in a badly crafted implementation.

A particular issue in the Hokkaido solution search is that an attempt involves very few computations (practically we will have $l = 26$), thus the checksum to validate a trial cannot involve real

Log of domain size $log_2(\|D\|)$	20	21	22	23	24
Arbitrary inverse (MB)	5.0	10.5	22.0	46.0	96.0
Permutation (MB)	2.5	5.3	11.0	23.0	48.0

Table 1: Compact bit-aligned storage sizes for domain sizes

cryptographic-strong hash function computations every few cache misses!

Moreover, whatever the checksum precision, there is always the possibility of unexpected collisions, so that several answers match. The choice of the problem instance by the server only insures that at least one such answer exists, but does not insure its uniqueness. Thus the server should really accept any starting point and binary path which leads to the target while achieving the expected checksum.

The choice of the checksum should thus both target a very fast computation, while still being discriminant enough so that not too many matching starting points and binary paths can be found. When testing one reverse path, the probability that it matches a non biased checksum only depends on the checksum size. As our purpose is mainly to ensure some level of work, the checksum size must be larger than the expected effort so that it does not interfere by providing other acceptable solutions too quickly.

With regard to the effort required by our scheme in practical cases, a simple and fast 32-bits integer checksum, which can be computed on the fly in a recursive implementation with very few operations, is both attractive and sufficient: *e.g.* $c = \bigotimes_{i=0}^{l} \mathrm{rot}(x_i, i)$. In order to balance the checksum speed and precision, a second level stronger checksum could be added, but it was not necessary with our chosen parameters during our experiments.

Tabulated data structures

Following related work, the size of the data structures needed for the tabulated function, whether f^{-1} or f for the forward variant, must be much larger than the expected cache size on high-end hardware but small enough to fit in the main memory of low-end machines. A minimal value of 16 MB is considered appropriate as of 2003's technology.

The storage is trickier an issue than it would seem: the compact bit-aligned approach has a significant computational cost to deal with address translations and value alignments. A non-compact storage requires less computation, but at the price of unused memory. If we restrict ourselves to power of two sizes for the domain $n = 2^N$ and compact storages, the actual usable sizes are shown in Table 1. If no power of two sizes are chosen, then restriction operations for index memory access will result in integer modulo operations, not welcome on microprocessors. Basically, the storage scheme involves memory to computation tradeoffs: the smaller the available memory, the harder the computations. Low-end machines may have to pay twice...

An overall reasonable value for the domain size is $N = 22$ and was used for our experiments. The length parameter can be chosen pretty independently of the domain size, provided that enough accesses are performed in the table, which leads to the constraint that $l > N$. Otherwise, why bother building such a large table? Also, the table building should be somehow a small part of the response computation.

4 One-Way Hokkaido

This section describes one-way variants of the Hokkaido protocol and discusses various choices of parameters.

4.1 Solution-verification protocols

The first variant is simply an adaptation of the forward challenge-response Hokkaido version described above. It aims at computing a stamp for a message which will be very fast to verify. It goes as follows:

Let f be a tabulated permutation in Domain D. Let x_0, n_i and m_i for $1 \leq i \leq l$ elements in D somehow derived from the message or service. Elements n_i and m_i must be different so that the x_{i+1} value will always differ depending on b_i. Then the client must find a binary path b_i of length l so that with:

$$x_{i+1} = f(x_i) \otimes (\text{if } b_i \text{ then } n_i \text{ else } m_i)$$

some low probable property holds on the checksum of sequence $x_{i,0 \leq i \leq l}$.

In this variant, two small mangle arrays n_i and m_i are used: with a single mangle array, the client could seek a solution without memory accesses by varying the messages to find an x_0 that would satisfy the property with an easy precomputed weak fixed 0 binary path in f.

The same issues as discussed in the previous section are raised by this variant: the checksum computation must be very cheap and there is a memory-computation tradeoff for compact bit-aligned data structures. The next variant uses a simple integer table for f as in Dwork *et al.*

Let t be a tabulated function from Domain D to computer integers. Let x_0, n_i and m_i for $1 \leq i \leq l$ be computer integers derived from the message or the service. Let r be a restriction function from any integer to D, for instance a modulo or a mask. Then the client must find a binary path b_i so that:

$$x_{i+1} = x_i \otimes t(r(x_i)) \otimes (\text{if } b_i \text{ then } n_i \text{ else } m_i)$$

and some low-probable property holds for x_l. Elements n_i and m_i must not be neutrals or identical for $r(x \otimes n_i)$ so that x_{i+1} always differ depending on b_i. A possible very cheap checksum for the sequence in this integer table variant is to chose the end point x_l as the checksum, if it is large enough, depending on the chosen parameters.

4.2 Discussion

This section discusses some choice of parameter values for setting the search effort, and how to build an interesting tabulated function.

The size of the search space is driven by parameter l, but the effort spent in the search is only based on the chosen probability of the property, typically that w particular bits of the checksum are all equal to 0. For an expected 2^w effort, a parameter $l = w + 10$ can be chosen so that the probability not to find a solution is as low as $e^{-2^{10}} = e^{-1024}$.

In the integer table variant, the target property must not interact with r: if the restriction function takes right bits then the property must consider those on the left. Otherwise the computation of part of the values would not be needed.

An Internet electronic mail is composed of several parts: (1) the SMTP [9] *(Simple Mail Transfer Protocol)* envelope which specifies the recipients of the message; (2) the mail headers, some fields of which are fixed by the sender user agent such as `Subject`, `From`, `To`, `Date` and others by MTAs such as `Received` fields; (3) the actual message contents, whether a bill or a love letter. Note that the recipient may legitimately not appear in the headers when *background carbon copies* are used.

A key idea is that a stamp should be paid for every recipient, so the solution seeking process above should be performed for every recipient email addresses. It should also vary for all message contents and user specified headers: for instance, the same stamp should not be reusable to send the same contents with different subject lines.

From the server point of view, a key point is that the tabulated function should not be recomputed over and over. On the other hand it may be wished that the function differs from server to server. A solution to this problem is to make a pseudo-randomly built tabulated function to depends on a per mail domain seed. Namely, if the target recipient is `hobbes@comics.net`, then the seed should depend on the `comics.net` domain name only.

The seed may be computed from a cryptographic hash of the mail domain name. However, this would fix the function as well as its domain size permanently. That would not allow different policies to be implemented on different servers. It would set an excessive burden on servers which have to handle a great number of domains. The next idea is to make yet another use the name server [8] infrastructure to publish this information for a given domain, as it is already used for mail exchangers (MX) or black lists [20]. For instance, `estamp.comics.net` could point to some pseudo *cname* that would provide the expected effort, pseudo-random generator seed, function variant and other parameters needed to compute a stamp. The answer could look like: `seed-0x4a2f107.size-22.effort-26.estamp`.

As for x_0, n_i and m_i integers, they can be derived from a pseudo-random sequence seeded by the hash of the message contents and client headers, or of the service description.

5 Performances

Memory-bound schemes such as those presented here or in related work are necessarily bound by memory bandwidth. Indeed, every approach involves a seeking process performed on a large search space with few computations and many memory accesses to make it memory-bound. As few computations are needed, the processor is available for exploring other paths or trials when a cache miss occurs, generating the next cache miss before the results of the first one is returned. This creates an opportunity to perform computations during cache misses, hence overlapping computations and memory accesses.

This overlapping can be achieved manually with a carefully designed parallel implementation of the functions which performs preloads of data ahead of their use. We have developed such a fast implementation for Dwork *et al*'s *mbound* function, the results of which are presented in the following. Such overlapping and better performance can also be achieved automatically if the processor architecture allows parallel computations to be performed when a cache miss occurs, as with hyper-threaded processors. In such a case, launching parallel search threads or processes allows a faster exploration if more memory bandwidth is available. Thus the random access memory bandwidth ultimately limits the seeking process, and not the previously claimed memory latency. They are both characteristics of the hardware, but the memory bandwidth is not as flat as the latency from low-end to high-end machines.

Identifier	A	B	C
cpu type	P2	P4 M	P4 HT
cpu freq. (MHz)	310	1200	3000
cache size (KB)	512	2048	1024
mem. lat. (ns)	285.0	180.0	125.0
mem. bw. (Mline/s)	6.5	14	43
rel cpu freq.	1.0	3.9	9.7
rel mem. lat.	1.0	1.6	2.3
rel mem. bw.	1.0	2.2	6.6

Table 2: Tested machines

Description	A	B	C
$N = 10$ in cache (ns)	89.7	21.2	11.0
$N = 22$ naïve (ns)	314.2	173.1	137.0
$N = 22$ fastest (ns)	242.5	121.5	51.0
naïve slowdown	3.5	8.2	12.5
fast slowdown	2.7	5.7	4.6
rel in cache	1.0	4.2	8.2
rel naïve	1.0	1.8	2.3
rel fastest	1.0	2.0	4.8

Table 3: Normalized mbound performance and comparisons

We now present experimental results on various computer architectures with a one-way Hokkaido variant and the *mbound* function of Dwork *et al.* [3]. We have written fast parametric implementations for the algorithms with great emphasis on optimizations: parameters are compile-time constants, plus compiler preload and branch predictions hints are provided when available. Non-compact data structures are used.

Table 2 shows some machines used in our experiments. Their capabilities vary greatly. The middle rows display raw figures for various hardware features, and the bottom rows show relative comparisons wrt the smallest machine. Machine C cpu is 9.7 times faster than machine A, although its memory latency is only 2.3 times faster. The memory latency was measures with some code of our making. The bandwidth is a theoretical figure. It focuses on cache lines per seconds as we are interested in random access performances. The relative figures in the three lower rows synthesize the hardware improvements over few years which is large on the frequency, small on memory latency and average on memory bandwidth.

Table 3 shows performance figures of our implementation of the Dwork *et al.*'s *mbound* function. The figures are normalized in nanoseconds by dividing the total time of the search by the actual number of table accesses. The slowdown figures measure how the performance is affected by main memory accesses compared to in-cache behavior. The relative figures allows to compare the different machines. For a small table, all data are in cache and the relative performance of the machine is closely correlated with the raw cpu frequency presented in Table 2. For large tables, two results are presented. The first one is an optimized naïve implementation which is intrinsically bound by the memory latency. It reproduces the experimental results obtained by Dwork *et al.* The second one

Description	A	B	C
$N = 10$ in cache (ns)	159.8	36.1	15.0
$N = 22$ naïve (ns)	387.8	135.0	136.6
$N = 22$ best (ns)	354.0	135.0	83.3
naïve slowdown	2.4	3.7	9.1
best slowdown	2.2	3.7	5.6
rel. in cache	1.0	4.4	10.7
rel. naïve	1.0	2.9	2.8
rel. best	1.0	2.6	4.2

Table 4: Normalized one-way Hokkaido performance and comparisons

Scheme	*Challenge/Verification*		*Work ratio*	
	cost	type	normalized	
Abadi *et al.*	15000	comp.	22K	$\approx 2^{14.5}$
Dwork *et al.*	2048	mem.	32K	$= 2^{15}$
Forward Hokkaido	26	mem.	2.6M	$\approx 2^{21.3}$
Hokkaido	26	comp.	13M	$\approx 2^{23.6}$

Table 5: Comparison for $2^{26} = 64M$ table memory accesses

is the best performance achieved by overlapping computation and memory accesses in our parallel optimized implementation, designed to hide the memory latency. The search loop is unrolled and jammed, and hyper-threading features are used when available. The slowdown wrt the in-cache version is significantly reduced, and the overall performances clearly outperform the memory latency, as Machine C is 4.8 times faster than Machine A with the parallel implementation. Another interesting figure is that the parallel implementation provides more than a two-fold improvement on the better hardware by going from 137 down to 51 ns, although the gain is much limited on the smaller one.

Table 4 outlines the performance of our one-way Hokkaido integer table variant. The results are very similar to those obtained with *mbound*. As expected, the in-cache performances are correlated to raw cpu, the naïve implementation is close to memory latency and the best performance is tight to memory bandwidth. Not as much effort on the optimization could be done for this implementation, as unroll-and-jam transformations can not be performed very easily on our recursive search function.

Finally Table 5 compares the different memory-bound schemes for a response or solution work of 2^{26} table accesses targeting a proof a 10 seconds work. The comparison is rough and must be taken with caution, as different types of protocols are compared and it is not based on actual experiments, but on the expected number of memory accesses for a given set of parameter values. For normalizing the solution-to-verification or response-to-challenge work ratio, a table access is considered 5 times worth a computation. Abadi *et al.* and Dwork *et al.* show quite close client-to-server work ratio results for the work involved because the later scheme, although consuming less operations, involves more expensive table memory accesses in the verification cost. The figures for our Hokkaido variants show much higher work ratio, thanks to their exponential behaviors. A very low 26 cost for the challenge or verification part is sufficient to achieve the expected work on the other side.

6 Conclusion

Following Dwork and Naor [4] idea to use proof of work functions to limit the rate of denial-of-service attacks, and Abadi *et al.* [1] idea to rely on memory-bound functions in such schemes so as to reduce the influence of Moore's law when computation-bound functions are chosen, we presented new proof of work memory-bound functions together with experimental results.

Our functions include both interactive challenge-response and one-way solution-verification variants. As related work, they rely on large tabulated or tabulated inverse of functions, possibly permutations, to require slow out-of-cache pseudo-random memory accesses. The amount of work for computing a response is exponential with respect to the work needed to set the challenge.

The results obtained by our different variants outperform previously obtained polynomial client-to-server work ratio. This exponential behavior is achieved by pseudo-randomly mangling a path through a tabulated function. Code optimizations involving data structures and parallelism transformations have also been discussed. Experiments and analyses show that such memory-bound schemes are bound by memory random access bandwidth and not, as previously thought, memory latency.

In order to make proof of work memory-bound schemes a workable solution to the particular spam problem, a range of issues must be addressed and solved.

First, there is a technical issue, as these schemes are bound by memory bandwidth, which depends on machine price and design date. As a result, such schemes do not fulfill their promises as high-end hardware often offer significantly better memory bandwidth performance than low-end machines. Also, on the low-end of the hardware spectrum, where small machines run primitive operating systems, the impact of the stamping computation may be significant on other user applications and hamper the machine's usability.

Second, on the practical front, the implementation of such a scheme requires a standard to be agreed upon and deployed on both client and server sides. There are many clients, and how to deal with the unavoidable transition period is unclear.

Third, on the economical side, more expensive mails are paid by everybody, whether spammers or not: this hurts the common sense of justice. As far as the spamming problem is concerned, server-to-server interactions as well as list servers should not be penalized by such a scheme. Some other approach should be required for these, such as building a trusted network. Indeed, if such scheme were applied to list servers, stamps should not be paid per recipient. However, if they were paid per mail independently of the recipient, this would make an easy loophole for spammers to send the same garbage to many people.

Finally, an interesting open issue is whether the effort involved in one-way solution-verification schemes could be made effort-bound. All proposed techniques, including our own presented here, rely on a probabilistic partial collision search which displays high variance efforts, including unlucky instances where the client can be stuck for a long time.

Acknowledgment

Thanks to Pierre Jouvelot, François Irigoin, Sebastian Pop, Corinne Ancourt and others for their help in improving the contents, the structure, the data and the wording in this paper.

References

[1] Martín Abadi, Mike Burrows, Mark Manasse, and Ted Wobber. Moderately hard, memory-bound functions. *ACM Trans. Inter. Tech.*, 5(2):299–327, 2005. A previous version appeared in NDSS'2003.

[2] Adam Back. Hashcash package first announced. `http://www.hashcash.org/papers/announce.txt`, March 1997.

[3] Cynthia Dwork, Andrew Goldberg, and Moni Naor. On memory-bound functions for fighting spam. In *Advances in Cryptology — CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 426–444. Springer, 2003.

[4] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology—CRYPTO '92*, pages 139–147. Springer, 1992.

[5] Benjamin Franklin. Advice to a young tradesman, 1748.

[6] Matthew K. Franklin and Dahlia Malkhi. Auditable metering with lightweight security. In *Financial Cryptography 97*, 1997. Updated version May 4, 1998.

[7] John L. Hennessy and David A. Patterson. *Computer Architecture, a Quantitative Approach.* Morgan Kaufmann Publishers, third edition, 2003.

[8] IETF. RFC 1035, domain names - implementation and specification. `http://www.ietf.org/rfc/rfc1035.txt`, November 1987.

[9] IETF. RFC 2821, simple mail transfer protocol (SMTP). `http://www.ietf.org/rfc/rfc2821.txt`, April 2001.

[10] Markus Jakobsson and Ari Juels. Proofs of work and bread pudding protocols. In *Comms and Multimedia Security 99*, 1999.

[11] Paul Judge. Taxonomy of anti-spam systems. `http://asrg.sp.am/`, March 2003. draft, version 3.

[12] Ari Juels and John Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. In *NDSS 99*, 1999.

[13] Ben Laurie and Richard Clayton. "proof-of-work" proves not to work. In *WEAS 04*, May 2004.

[14] José-Marcio Martins da Cruz. Mail filtering on medium/huge mail servers with j-chkmail. In *TERENA Networking Conference 2005*, Poznań, Poland, June 2005.

[15] MessageLabs. Spam intercepts. `http://www.messagelabs.com/`, 2005.

[16] Monty Python. Spam skit. Flying Circus episode 25 (season 2), broadcast on BBC One, december 15 1970.

[17] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[18] Ron Rivest and Adi Shamir. Payword and micromint – two simple micropayment schemes. *CryptoBytes*, 2(1), 1996.

[19] David S. H. Rosenthal, Mema Roussopoulos, Petros Maniatis, and Mary Baker. Economic measures to resist attacks on a peer-to-peer network. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[20] Paul Vixie. DNSBL – DNS-based blackhole list. part of MAPS, Mail Abuse Prevention System, 1997.

[21] XiaoFeng Wang and Michael Reiter. Defending against denial-of-service attacks with puzzle auctions. In *IEEE Symposium on Security and Privacy 03*, May 2003.

[22] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

Typeset with LaTeX, document revision 266