Exponential Memory-Bound Functions for Proof of Work Protocols

Technical Report A/370/CRI version 3

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

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 deter such phenomena and other denial-of-service attacks by requiring *computed* stamps based on hard-to-solve problems with easy-to-verify solutions. As cpu-intensive computations are badly hit over time by Moore's law, memory-bound computations have been suggested to deal with heterogeneous hardware. We introduce new practical, optimal, proven and possibly memory-bound functions suitable to these protocols, in which the client-side work to compute the response is intrinsically exponential with respect to the server-side work needed to set or check the challenge. One-way non-interactive solution-verification variants are also presented. Although simple implementations of such functions are bound by memory latency, optimized versions are shown to be bound by memory bandwidth instead.

Keywords: proof of work protocol, memory-bound function, anti-spam technique.

1 Introduction

In recent years, the Internet electronic mail user has been plagued with massively sent unsolicited messages, or spams [17]. This communication channel has proven interesting to marketers and crooks thanks to its combined worldwide-spread use in upscale households and very low sender-cost per message: two thirds of all emails were spams in 2005 [16]. This phenomenon can be tackled by preventing, deterring, detecting or responding to it appropriately [12]. Behavior and contents filters [15] allow organizations to reduce the user and system burdens of these messages.

We focus here on a cryptographic technique by Dwork and Naor [6] which aims at putting a tighter economic bound to spamming by making emails more expensive to send, thanks to stamps. The 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* [7]. In the Hashcash [3] scheme, a stamp is built for a service, such as sending a mail to an address today, by producing a bit-string: the hash of the bit-string and of the service description must start with a number of leading zeros. The high solution cost comes from enumerating many bit-strings up to one having the partial hash collision property, but the verification is one straightforward computation on the provided solution.

Economical measures to contain denial-of-service attacks have been pursued for other purposes than deterring spams: proof of work can introduce delays [19], help audit reported metering of web-sites [8] or make a digital data preservation protocol resistant to malign peers [20]; puzzle resolutions [13] or auctions [23] are used to limit the incoming flow of service requests; [11] formalizes proof of work schemes; finally, actual financial analysis are suggested as useful [14] 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. A key issue in both approaches is to compare the effort required of the client to compute the stamp in the response or solution part with the work of the server to set the challenge or verify the solution.

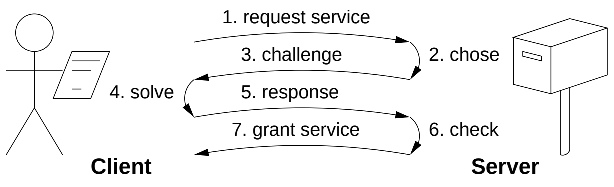

Figure 1: Interactive Challenge-Response Protocol

The challenge-response approach 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* 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 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.

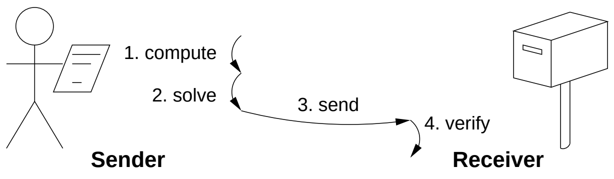

Figure 2: One-way Solution-Verification Protocol

The solution-verification approach targets asynchronous off-line message composition and later transfer. It does not need any interaction between the sender requesting the service and the final service provider. As the problem to solve is self-imposed, it depends somehow on the requested service, and the receiver must validate both the chosen problem and the provided solution.

Processor computational performance varies more widely than cache to memory access performance [24] from high-end servers to low-end personal digital assistants and over time, following Moore's law [18]. Thus, Abadi *et al.* [1] suggests 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 novel approach is further investigated by Dwork *et al.* [5].

This paper presents new contributions about memory-bound proof of work functions. Section 2 presents and analyzes related work by Abadi *et al.* and Dwork *et al.* Section 3 describes Hokkaido, our new challenge-response protocol for memory-bound proof of work schemes. For a challenge cost of $\mathcal{O}(\ell)$ the response cost is $\mathcal{O}(2^\ell)$ memory accesses, thus inducing an optimal exponential work for the client compared to the server. Section 4 details one-way variants for the same purpose. Section 5 contributes experimental results. It shows that optimized implementations of these functions are bound by memory bandwidth, not by memory latency as previously believed. 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 asks the client wanting to send it to perform a computation that requires $\mathcal{O}(\ell^2)$ memory accesses, although the verification of computation result costs $\mathcal{O}(\ell)$, where ℓ is a length parameter. The challenge-setting phase uses a sequence x_i of $\ell + 1$ elements starting from a chosen x_0:

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

where f is a random-like function on a domain of size 2^n and $\oplus$ the exclusive-or operator. The response looks for x_0 by computing a reverse path starting backwards from x_ℓ:

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

and checking the path against a provided checksum of size m. As f^{-1} is not a simple function, possible multiple pre-images at each stage may lead to multiple paths: The response-to-challenge work ratio comes from exploring 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. Assuming that Function f is known, the amount of communication involved in a protocol instance is about $n + m$ bits for the challenge and n bits for the response.

This novel technique has some drawbacks. First, the solution cost is *only* quadratic, requiring a sizeable amount of server verification if the client is to provide proof of significant work: a large value $\ell = 2^{13} = 8192$ is suggested in practice. Second, this quadratic behavior depends on the chosen function to be *random* and on the forward path to be *known* to exists: a permutation gives equal challenge and response works as only one reverse path exists; a random function without known forward path does not display the quadratic effect either as only one reverse path exists on average. Third, the actual multiplier hidden by the $\mathcal{O}()$ is a small $\frac{1}{2(e-1)} \approx 0.3$ for purely random functions: few reverse paths are found as 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 involve a computation-memory trade-off: either fast lookups need more memory, or a packed representation requires a higher computation cost, defeating the purpose of memory-bound functions as a better representation can be chosen for better hardware. Moreover, it must handle a variable number of pre-images, what requires a loop and tests which add to the computation overhead.

Dwork *et al.* [5] propose a non-interactive abstract and proven one-way scheme, plus a concrete instantiation inspired by the RC4 cipher. Solution-seeking trials are performed 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 μ and k:

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

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 $h(s_\ell)$. Verifying a solution requires to perform the full trial computation again for the provided parameter k. The communication cost involved is about $\log(E)$ bits.

The verification costs $\mathcal{O}(\ell)$ for a $\mathcal{O}(E\,\ell)$ memory accesses exploration. Although a simple constant work ratio seems less interesting than the previous proposal, E can be chosen quite freely to tune the solution-seeking cost. However, even if setting $E = 2^\ell$ results in a theoretically exponential work ratio, it is not practical: a large $\ell = 2^{11} = 2048$ is needed to amortize the cryptographic-strong hash computations involved (for instance, SHA1 requires about 1000 operations per block), as well as insuring that the internal state is used thoroughly. It would thus induce a tremendous amount of work if used to achieve the exponential ratio: Nobody want to perform 2^{2048} anything. A more realistic 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 four 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. Moreover, we show that the expected time to compute a proof of work depends on the more variable memory bandwidth, rather than the memory latency previously suggested by both related work.

3 The Hokkaido protocol

The forward and backward variants are described, before discussing parameters and security.

3.1 Challenge-response protocol

Let D be a finite integer domain of size 2^n. Let ℓ be the (will-be short) path length. For every $1 \leq i \leq \ell$, let f_i and g_i be $D \rightarrow D$ functions, discussed further in the next section. Let h be a checksum function from any sequence of D elements into a finite integer domain of size 2^m. The server chooses a starting point $x_0 \in D$ and a binary path b of length ℓ. Then it computes ℓ mangling iterations starting from x_0, using every two functions:

$$x_i = \text{if } b_i \text{ then } f_i(x_{i-1}) \text{ else } g_i(x_{i-1}) \tag{1}$$

Finally the checksum of the path from x_0 to x_ℓ is computed as $c = h(x_0 \ldots x_\ell)$. The challenge is composed of ℓ, h, for every $1 \leq i \leq \ell$ f_i and g_i, x_0 and c. The response is a binary path b matching the checksum through Formula (1). It may be different from the server-chosen binary path, depending on checksum collisions.

A first worst-case $\mathcal{O}(2^{n\ell})$ search algorithm is to try all sequences without even using the mangling functions, and to check the validity of the found hash matches by rebuilding the binary path. A better worst-case $\mathcal{O}(2^\ell)$ algorithm is to enumerate all possible paths starting from x_0 through Formula (1) with the mangling functions. If the function computations require memory accesses, *e.g.* they are tabulated, the search can be memory-bound.

As Abadi *et al.*, the challenge is based on forward computations, but here the response is also forward: the reverse-path computation trick is not needed to induce a significant work ratio, as it is already intrinsically exponential thanks to the binary path mangling. Nevertheless, a backward variant can be devised simply by giving the end point x_ℓ in the challenge, and requesting both x_0 and the binary path b as the response. The starting point is needed by the server to check the possibly different client solution. This variant could also benefit from the fact outlined by Abadi *et al.* that computed forward functions may require tabulated inverses, adding to the asymmetry of the workload.

3.2 Discussion

Let us now discuss the choices of the various parameters in the above algorithm, namely the mangling functions, domain size, path length and checksum.

Mangling functions

Formula (1) requires 2ℓ tabulated functions in D. Random functions are not very interesting because of the collisions, which reduce the effective number of path. Instead, we want to rely on collision-free random permutations. However, having to build and handle 2ℓ tabulated permutations is troublesome for both server and clients: we would like just one large table accessed randomly enough to make the search process memory bound.

Thus we propose to build these 2ℓ permutations as follows: Let t be a tabulated permutation in D. Let us choose 2ℓ distinct integers in D: v_i and w_i for $1 \leq i \leq \ell$. Let $\oplus$ be the exclusive-or operator. For the forward variant, the 2ℓ mangling functions are defined as:

$$\forall x \in D, f_i(x) = t(x \oplus v_i) \text{ and } g_i(x) = t(x \oplus w_i) \qquad (2)$$

For the backward variant, better mangle *after* the table access. These functions are permutations because composed of permutations, the distinct integers make them collision-free, and their values can be easily computed from t and a cheap xor. The xor-mangling is not motivated by the short cycle issue mentioned by Abadi *et al.* which may be used to help the search in their scheme, because such cycles are unlikely with permutations followed on a small path length. However it is necessary for the functions to be near-independent, and thus insure that memory accesses are needed.

Tabulated permutation t must be known to both client and server. It must not be computable, so that memory accesses are needed. It could be built from a random generator; however its distribution would require to transmit about $n2^n$ bits. Another option is to build it from a known pseudo-random generator, so that only the seed is needed. Building such a table requires about 2^n random memory accesses. This is fine on the client, which is expected to perform many memory accesses, but quite annoying on the server. Thus we propose that when a server builds a table it keeps it long enough so as to amortize its building cost on many challenges. The 2ℓ distinct integers used to build the functions could also be derived from the shared pseudo-random generator. They can also be generated quite cheaply by the server for each challenge. Finally, the checksum function is assumed to be fixed by the protocol. Under these arrangements, the challenge sends $s+2\ell n+n+m$ bits for the seed, integers, starting point and checksum, and the response only ℓ bits.

Domain size and path length

Following related work, the size of the tabulated function must be much larger than the cache size on high-end hardware but small enough to fit in the main memory of low-end machines: about 16 MB is considered appropriate as of 2003's technology. Moreover, the client delay must be both significant and reasonable, which is achieved with about 64 million memory accesses.

A compact tabulated permutation in D needs $n2^n$ bits, at the cost of shift and mask operations to deal with memory alignments: $n = 22$ and 23 lead to 11 and 23 MB respectively. A simpler word-aligned data structure is less economical in memory and cache miss rate.

The length parameter can be chosen pretty independently of the domain size, although $\ell > n$ seems reasonable so that the table is heavily used and its building is negligible for the client. Targeting 2^{26} memory accesses for the client, our experiments use $n = 22$ and $\ell = 26$, that is a 11 MB table and a mere 26 memory accesses challenge or verification on the server, plus the amortized table-initialization cost.

Checksum function

The last parameter is the checksum function to be computed on every path. Its size must be large enough so that the probability of finding a different matching path is low, and does not interfere with the overall search complexity. Thus we require $m > \ell$. With our above settings, $m = 32$ is sufficient as it induces a small collision probability of about $2^{\ell-m} = 2^{-6} \approx 1.5\%$.

As there are a lot of trials, each one involving very few operations, a cryptographic-strong software function cannot be used, as such functions are usually cpu-intensive, thus would break the short-path memory-bound property. The best choice is a strong but fast hardwired checksum on the processor, if available. As a second-choice low-cost software checksum, a simplistic idea is to return the m last bits of the path. However, this gives away part of the answer and allows to shorten the search: some mangling is necessary. A simple yet convenient proposition is $c = \bigoplus_{i=0}^{\ell} \mathrm{rot}(x_i, i)$ where $\mathrm{rot}(x, i)$ is the bitwise rotation of integer x by i bits. As it is cumulative, it can be computed on the fly incrementally in a recursive search.

3.3 Hokkaido Security

This section proves the Hokkaido function security under ideal assumptions, by showing that a *best* search algorithm requires in the *worst-case* 2^ℓ checksums, 2^ℓ memory accesses, and $\mathcal{O}(2^\ell)$ memory fetches. It also describes an attack that reaches the provided memory fetches lower-bound, as well as a simple counter-measure. All complexities are worst-case unless otherwise stated. The actual detailed proof is presented in the Appendix.

Proof Sketch

We first define a valid path (Definition 1) and perfect hash function (Definition 2). Lemma 1 counts the number of valid paths and Lemma 2 states that all such paths may be enumerated by an iterative search. Then Theorem 1 shows that in the worst-case a valid path search can involve an exponential number of computations. Theorem 2 proves that this work ratio is optimal for interactive challenge-response protocols. One-way solution-verification protocols rely on finding an item matching a probabilistic property in a possibly infinite search space, thus their work ratio is probabilistically bound, without absolute upper-bound, and is not related to the search-space size but to the probability of the property. This first stage of the proof does not address whether the search is memory-bound, but simply focus on the computation-bound aspect.

We then proceed to define a *best* search algorithm (Definition 3), which displays this worst-case optimal complexity, and only focus on valid paths (Lemma 3). This is paradoxically a subtle restriction, as we neglect by doing so random searches. For such non-best algorithms, it is indeed possible to have a far worse computation complexity, but a much better memory access complexity for some settings: For the $2^{n\ell}$ sequence enumeration algorithm in Section 3.1, which does not use the mangling functions in the enumeration, and given a sufficient checksum size m, the search may only need to verify that one found matching path is valid, requiring only about 2ℓ memory accesses. However, such solutions, even with hybrid valid-path enumeration parts, seem impractical because of their computation complexity. Our valid-path $\mathcal{O}(2^\ell)$ enumeration search described in Section 3.1 is a *best* algorithm.

As a third part of the proof, we discuss collision-free independent functions or permutations, defined as near-independent (Definition 4). Near-independence is independence slightly skewed so as to be collision-free. These functions result from our construction of f_i and g_i functions based on a single table (Lemma 4). Pretty-independence is defined (Definition 5) as near-independence or something closer to independence, allowing possible low-probability collisions. Some useful properties are derived on these functions (Lemma 5 and Lemma 6). Values computed from near- or pretty-independent functions will be described as near- or pretty-independent one from the other, meaning that nothing is known about them apart that they may be distinct.

Then we show through somehow technical Lemma 7 and Lemma 8 that distinct binary paths implied by a search result in pretty-independent path values. This is central to our demonstration, because these pretty-independent values will result directly in memory accesses. It also suggests that search algorithms that would take advantage of the search history by memoizing some results do not avoid these memory accesses, as they involve pretty-independent values anyway.

We then define memory accesses as a unit of cost (Definition 6), incurred from computing the value of a random function or permutation on a value pretty-independent from previously available data (Axiom 1). This cost applies to our mangling functions (Lemma 9). The idea behind this axiom is that a tabulated random function must always generate memory accesses to get a value. From a theoretical point of view, this is not true. For instance, an implementation may memoize recently accessed values in registers and check whether a value is reused. However, there are few registers thus hits are unlikely.

The second simplifying trick is that we also deal with non deterministic algorithm that would memoize intermediate results by stating that memoization basically will also cost a memory access (Axiom 2). This is not absolutely true either. Instead of using some data-structure to check whether a binary-path was already visited, a non-deterministic algorithm could be expanded, dropping loops and calls, so that computed set information is implicit from the point in the execution. However, this would lead to a very large code, and although no data structure would be used, the memory accesses would be paid for accessing the code instructions.

Based on these two axioms, we can now demonstrate our main result, namely Theorem 3 which states that an exponential number of memory accesses is necessary to find a solution, whatever the best algorithm, whether deterministic or non-deterministic in its enumeration. Our valid-path 2^ℓ enumeration search reaches this bound.

We then proceed likewise to compute a bound on memory fetches (Definition 7), which is also axiomatized (Axiom 3) and then demonstrated through Lemma 10 and Theorem 4. A cache line is fetched from the main memory, bringing data to the processor. There may be no actual transfer if the line is already in the cache: what is meant by a memory fetch is a *potential* cache miss because unforeseen data are requested. Our approach imply the definition of related values (Definition 8) which depend one from the other through f_i or g_i functions, because such related values may be stored together to help the search. As a line is fetched for some x, all the coming data may be related to this x, if a careful arrangement of data has been devised by the algorithm. The amount of useful data is nevertheless limited by the cache line: if more data are to be accessed, several independent fetches are necessary.

In the last part of this demonstration, we define the coverage of the domain reached at each stage by a search (Definition 9) and compute it in Theorem 5, so as to check that most values are actually used: it diverges quickly on the first iterations then converges quickly towards one. Finally we discuss in Theorem 6 the impact of these various results on our concrete proposal, by computing the average complexity which can be expected with the Hokkaido search.

An attack to reach the lower-bound

The proof of Theorem 4 gives a clue about how to reach the provided worst-case lower-bound. The algorithm must organize its enumeration and data storage so that known to be needed related independent values are stored in one cache line. It also implies that a pattern of reuse is predictable. This is indeed the case, as from a given x_i through Formula (1), the very same paths are always followed. This reuse is likely to occur when the coverage is close to 1, typically for $i \geq n$ as shown by Theorem 5. A clever algorithm can store related values on the fly such as $t(x_n \oplus v_{n+1})$, $t(x_n \oplus w_{n+1})$ and so on which are related to a given x_n, and reuse them when available. In order to break these patterns, the mangling must change depending on the previous path, so that when arriving at an already encountered x_i, different paths are to be followed. This can be achieved cheaply by mangling the xor-permutation integer with something which depends on the previous path, for instance $v'_{i+1} = v_{i+1} \oplus h(x_0 \ldots x_{i-1})$ or even $v'_{i+1} = v_{i+1} \oplus x_{i-1}$, and *idem* for w'_{i+1}.

Proof comparisons

Our and Dwork *et al.* proofs are somehow similar yet complementary. In both cases, the proofs rely on the structure of the proposed functions. However [5] assumes in its Lemma 1 that the computations are necessarily needed to find a solution, and proceeds in its technical proof for Lemma 3 to show that it results in cache misses through a precise mathematical modelization of a computer memory hierarchy. In contrast, we roughly assume that somehow independent computations using tabulated function cost *memory accesses*, whatever that may be, simplifying this part of the proof a lot by merely axiomatizing it, and we rather focus on showing that the accesses are indeed independent and that there is no better search algorithm to find a solution, which leads us to characterize these algorithms and their deterministic nature. It is possible that our results would still hold on the whole if random functions are used instead of random permutations, but such proof would be more technical as it has to deal with collisions which may reduce the overall search complexity.

4 One-Way Hokkaido

This section describes one-way variants, the first directly adapted from the forward challenge-response protocol, the other relying on a simpler integer array, and discusses choices of parameters.

4.1 Solution-verification protocols

Let t be a tabulated permutation in D. Let ℓ be the binary path length. Let the client compute $2\ell + 1$ distinct integers in D, x_0, v_i and w_i derived from the specific message or service. The client must find a ℓ-length binary path b so that with:

$$x_i = t(x_{i-1} \oplus \text{if } b_i \text{ then } v_i \text{ else } w_i) \tag{3}$$

some low-probability property holds on the path checksum $h(x_0 \ldots x_\ell)$.

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 t, as Dwork *et al.*

Let t be a tabulated function from Domain D to words (typically 32 bits integers). Let the client derive $2\ell + 1$ distinct words x_0, v_i and w_i from the service. Let r be a restriction function from any word to D, for instance $r(x) = (2^n - 1)\text{and}(\text{rot}(x, n - 32) \oplus x)$, so that table accesses will always be performed on valid indices. The client must find a ℓ-length binary path b so that with:

$$x_i = t(r(x_{i-1} \oplus \text{if } b_i \text{ then } v_i \text{ else } w_i)) \tag{4}$$

some low-probability property holds for $h(x_0 \ldots x_\ell)$.

4.2 Discussion and security

First, $2\ell+1$ distinct integers must be derived from the message or service. This derivation should be expensive enough so that it is not worth trying to find other better values, for instance by varying the service description, but it must not be too expensive, as the server will have to verify the client choice. We follow related work and rely on a few cryptographic-strong hash

Identifier	**A**	**B**	**C**	**D**	*Unit*
cpu type	P2	P3	P6 M	P6 HT	Intel
cpu freq.	310	900	1200	3000	MHz
cache size	512	256	2048	1024	KB
mem. bw.	6.5	8	14	43	Ml/s
mem. lat.	285	147	130	125	ns

Table 1: Tested hardware

computations to mangle the description, so that the cost of deriving the integers is higher than performing a single trial, and that targeting specific values is ineffective.

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

In both protocols, the same table must be available to both client and server. As pointed out before, the computation of these structures must be a small part work, and the server should not have to do it over again. A first solution is to build a table pseudo-randomly with a per-service seed. For instance, if the recipient is `hobbes@comics.net`, the seed would be the hash of the domain name only. However, this would not allow servers to have their own policy. It would be very costly for mail servers that home many domains. Another idea is to distribute the needed parameters, such as seed, table size, required effort and so on by mean of the name server [10] infrastructure, as already used for mail exchangers or black lists [22].

The different theorems in Section 3.3 can be adapted to these solution-verification protocols. The key difference is that the amount of work is probabilistically defined, without actual upper-bound. Basically, a perfect checksum function hides the paths and thus many paths must be enumerated. As the involved functions are near-independent, their combinations result in many unrelated memory accessed, and many memory fetches. For a 2^{26} client effort, the server verification work involves $36 = 26 + 10$ memory accesses only, plus the verification of the chosen parameters.

5 Performances

Memory *latency* and *bandwidth* characterize processor to main memory transfer [9] performance. Latency is the time between a load instruction and the availability of the data. It is driven by the propagation of signals within the hardware: the physics [21] involved does not evolve much in time or for more expensive computers. Bandwidth is the amount of data that can be fetched from memory per unit of time. It can be improved by widening the transfer bus and its related logic so that more data are fetched together. On inexpensive hardware, these figures are somehow merged: the available bandwidth corresponds to the latency, *i.e.* one cache miss is handled at a time. Table 1 illustrates these facts about hardware used in our experiments. From **A** to **D**, cpu is about 10 times faster, bandwidth about 7 times larger, but latency only 2.3 times smaller. By multiplying bandwidth and latency, **A** may handle 2 concurrent cache misses, but **D** about 7.

Let us show that any memory-bound proof of work scheme is necessarily bound by memory bandwidth, if the search algorithm implementation is carefully crafted so as to parallelize independent memory accesses. Let us assume that several independent trials of the Dwork *et al.* algorithm are to be computed simultaneously on one processor: on the first table access of the first trial, the fetch is likely to induce a cache miss and thus a potential delay. However, the processor can switch to a second trial up to its first fetch, then switch to a third, and so

Function	*N*	*Description*	**A**	**B**	**C**	**D**
	22	sequential	314.2	199.6	173.1	137.0
mbound [5]	22	parallel	242.5	125.4	121.5	51.0
	10	in cache	89.7	43.5	21.2	11.0
	22	sequential	559.0	293.5	235.8	169.9
moderate [1]	22	parallel	556.0	293.5	231.0	110.0
	10	in cache	306.6	104.1	77.6	37.5
	22	sequential	387.8	228.8	135.0	136.6
hokkaido	22	parallel	354.0	217.5	135.0	83.3
	10	in cache	159.8	55.3	36.1	15.0
hardware native memory latency			285.0	147.0	130.0	125.0

Table 2: Apparent delay per memory access in nanoseconds

on. When finally coming back to the first trial computation, the very first requested data may be available, and the processor can resume its computation. Although each trial computation is indeed bound by memory latency, the overall optimized search is bound by memory bandwidth, as the latency is hidden by switching between trials.

The very same trick can be used on any other scheme, as they rely on finding one solution in a large search space. Each trial is necessary independent of others, because the server must be able to verify it without doing the whole search, thus a parallel search is always possible by doing it the way the server would check a solution. So the actual issue is to perform a parallel search. Processors can be *hyper-threaded*, thus can switch quickly between threads or processes on cache misses, or *dual-core*, where two cpus share the same memory. Launching parallel explorations on distinct parts of the search space takes advantage of such features. Moreover, special *preload* instructions can tell a processor to load data ahead of their use, to help hide this delay. They may be generated by the compiler or hinted manually. Finally, unroll-and-jam [2] loop transformations can suggest when it is best to switch from one trial to the next.

We have developed manually-tuned implementations [4] for Dwork *et al*'s *mbound*, as well as simple codes for Abadi *et al.*'s *moderate* and our Hokkaido function. Table 2 presents experimental results on various computer architectures with these functions. The figures are normalized in nanoseconds by computing the *apparent* latency of the memory accesses, *i.e.* the total search time divided by the number of accesses. For a small table, all data are *in cache* and performances are fast in all schemes. On large tables, the sequential *mbound* implementation reproduces Dwork *et al.* experimental results [5], with a delay intrinsically bound by memory latency. The optimized version is significantly faster: thanks to the high available bandwidth, **D** apparent delay is under half the real hardware latency. Taking this effect into account, we can restate the analysis of Section 6.2 in [5]: the expected time for computing a proof of effort for a memory-bound solution-verification function on a processor is $2^e \cdot \ell \cdot \frac{\nu}{b}$ with e the expected effort, ν the bandwidth, b the cache line size, where $\frac{\nu}{b} \leq \tau$ the memory latency. Less striking results are obtained for *moderate* and Hokkaido, because both functions rely on a recursive search which cannot be as easily optimized. Concurrent processes are used for parallel figures, but do not perform significantly better.

Finally, Table 3 compares the different memory-bound schemes for a fixed common response or solution work of 2^{26} table accesses targeting a realistic proof of 10 seconds work. The comparison is rough, as the two types of protocols are compared, and is based on the theoretical complexity of the computations. Solution-to-verification and response-to-challenge work ratios are normalized by arbitrarily assuming that a table access is worth 5 computation units, which is consistent with optimized versus in-cache figures in Table 2. Abadi *et al.* and Dwork *et al.* show

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

Table 3: Comparison for a $2^{26} = 64M$ memory accesses search

very close practical client-to-server work ratio. Our Hokkaido variants have a 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 [6] idea to use proof of work functions to deter 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, we have presented new proof of work functions together with experimental results. Our functions are practical, optimal, proven and possibly memory-bound. They include both interactive challenge-response and one-way solution-verification variants. As related work, they rely on tabulated functions 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, thanks to a simple binary path mangling. The communication overhead of our functions as well as those of the related work is low. Moreover, the initialization cost for the tabulated permutation on the server needed by some variants of our scheme is negligible if amortized on enough clients.

Although sequential implementations of such functions are bound by memory latency, optimized versions are shown to be bound by memory bandwidth instead, which varies more widely from low end to high end hardware. Even if this fact reduces the interest of memory-bound functions, our mangling path combinatorial technique is still interesting and the optimality results still hold for computation-bound functions based on our work.

In order to make proof of work memory-bound schemes a workable solution to the particular spam problem, a range of technical, practical and economical issues must be addressed. First, as these schemes are bound by memory bandwidth which depends on machine price and design date, they do not fulfill their promises as high end hardware offer large memory bandwidth. 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, their deployment requires a standard on both client and server sides. There are many clients, and how to deal with the unavoidable transition period is unclear. Third, 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 approaches should be required for these, such as building a trusted network. Indeed, if such schemes were applied to list servers, stamps cannot be paid per recipient. However, if they were paid per mail independently of the recipient, this would make an easy loophole for spammers.

Finally, an interesting open issue is whether the effort involved in one-way solution-verification schemes could be made effort-bound. Indeed, 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 bound in the probabilistic sense: in all such known methods, including our own, a user will statistically spend over 4 times the average effort every $e^4 \approx 55$ mails, hanging one's laptop for one minute once in a while. All proposed techniques rely on a probabilistic partial collision search which displays high variance efforts, including unlucky but not so infrequent instances where the client can be stuck for a long time.

Acknowledgment

Many 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. I am also in debt to reviewers of previous versions of this paper whose detailed comments, discussions and insights greatly helped to enhance the overall presentation and the justifications provided.

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] F. Allen and J. Cocke. *Design and Optimization of Compilers*, chapter A catalogue of optimizing transformations, pages 1–30. Prentice-Hall, 1972.

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

[4] Fabien Coelho. Implementation of memory-bound functions. `http://www.coelho.net/mbound.html`, September 2006. Optimized C source codes.

[5] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] 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.

[21] Ole Christensen Rømer. Démonstration touchant le mouvement de la lumière. *Journal des Sçavans*, pages 233–236, 7 décembre 1676.

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

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

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

Appendix

Here is the detailed proof of our scheme, which is discussed and commented in Section 3.3.

Exponential computations

Def 1 (valid path) *a sequence which follows Formula (1) for a binary path.*

Lem 1 *collision-free at each stage mangling functions in Formula (1) lead to* 2^ℓ *distinct valid paths.*

Proof There are 2^ℓ distinct binary paths of length ℓ. If two distinct binary paths lead to a same path, there is a first differing bit b_i with an identical x_i, thus $f_i(x_{i-1}) = g_i(x_{i-1})$, and the functions are not collision-free at stage i, contradicting the hypothesis. QED

Def 2 (perfect hash function) *behaves as a random function: its value on an entry must always be computed, and all values in the finite target space are equiprobable.*

Lem 2 (valid path search) *If h is perfect, any search for Formula (1) can involve up to the number of valid paths of h-operations to find a proven solution.*

Proof As h is perfect, it must be computed on any valid path to check if it matches. In the worst-case, only one valid path matches, and it is enumerated last of all by the search. QED

Theo 1 (exponential search) *If h is perfect, any search for Formula (1) with collision-free at each stage functions induces up to 2^ℓ h-operations to find a proven solution.*

Proof From Lemma 1 and Lemma 2. QED

Theo 2 (optimality) *The best work ratio of challenge-response protocols is the exponential.*

Proof The response to the challenge lies within a finite size σ space. Once found, it must be sent to the server to be checked, requiring about $\ln(\sigma)$ data communication and processing. QED

Best Search

Def 3 (best search algorithm) *has the above 2^ℓ upper-bound h-operation complexity.*

Lem 3 *A* best *algorithm only checks* valid *paths, at most* once.

Proof Otherwise it would break the worst-case upper-bound which defines a *best* search. QED

Independence

Def 4 (near-independence) *Functions f and g in D are near-independent iff they are collision-free $P(f(x) = g(x)) = 0$, but otherwise independent $P(f(x) = g(y)|x \neq y) = \frac{1}{2^n-1}$.*

Lem 4 *$\forall v \in D$, $v \neq 0$, $f(x) = x$ and $g(y) = y \oplus v$ are near-independent permutations in D.*

Proof Although the property seems obvious, a non-zero xor-permutations may be somehow skewed with respect to collisions... If $x = y \oplus v$ then $y \oplus x = y \oplus y \oplus v = 0 \oplus v = v$, *i.e.* $v = y \oplus x$ is a xor-permutation solution. Then if $x = y$, $v = y \oplus x = x \oplus x = 0$, contradicting the hypothesis, thus $P(x = y \oplus v|x = y) = 0$. For each x (2^n cases) and $y \neq x$ ($2^n - 1$ cases), $v = y \oplus x \neq 0$ is *the* solution for this x (2^n cases), hence $P(x = y \oplus v|x \neq y) = \frac{2^n}{2^n(2^n-1)} = \frac{1}{2^n-1}$. Admitting $v = 0$ would break near-independence as close to independence: the weak identity permutation would double the overall number of collisions. QED

Def 5 (pretty-independence) *functions at least near-independent, or closer to independence.*

Lem 5 (independence compositions) *Let f_1, f_2, f_3, f_4 be near-independent permutations in D. Let g be a permutation in D. Then (a) $f_1 \circ g$ and $f_2 \circ g$ are near-independent, (b) $g \circ f_1$ and $g \circ f_2$ are near-independent, (c) $f_1 \circ f_2$ and $f_3 \circ f_4$ are pretty-independent.*

Proof The first two properties are obvious because the common composition with one permutation does not change the collision probability. For the third property, one must compute the collision probability. QED

Lem 6 (application independence) *If f is a random function or a random permutation in D, then $\forall x \in D$, $f(x)$ and x are independent.*

Proof Compute $P(x = f(x)) = \frac{1}{2^n}$ in both cases. QED

Lem 7 (frontier) *If two binary paths b and b' have distinct i-length prefixes, then their x_i and x'_i values computed from Formula (1) for near-independent permutations are pretty-independent.*

Proof Let $j \leq i$ be the index of the first differing bit in the binary paths. If $j = i$ then through Formula (1) $x_i = f_i(x_{i-1})$ and $x'_i = g_i(x_{i-1})$ or vice versa, thus they are near-independent as f_i and g_i are near-independent. if $j < i$ then $x_j = f_j(x_{j-1})$ and $x'_j = g_j(x_{j-1})$ or vice versa. These intermediate values are near-independent. Following Formula (1) up to i, then by Lemma 5 either the cumulated permutations are the same, and the values are still near-independent at stage i, or the permutations differ somewhere, and the values are pretty-independent at stage i. QED

Lem 8 (horizon) *If two binary paths b and b' have distinct i-length prefixes, then the x'_i value computed for b' from Formula (1) with near-independent functions is pretty-independent of any x_j path variables corresponding to binary path b and of any x'_k with $k < i$ in its own path.*

Proof Same technique as Lemma 7: at least one near-independent function computation is involved to switch from one value to this x'_i. Within one path, values are independent by Lemma 6. QED

Exponential memory accesses

Def 6 (memory access) *a unit of cost for transferring some bits from memory to processor.*

Axiom 1 *computing $t(x)$, where t is a random function in D and x is pretty-independent from previously available x, costs* one *memory access implying the transfer of n bits.*

Lem 9 (mangling function cost) *computing $f_i(x)$ and $g_i(x)$ defined by (2) where x is pretty-independent from previous values costs* one *memory access for each computation.*

Proof $x \oplus v_i$ and $x \oplus w_i$ are near-independent by Lemma 4 with $v = v_i \oplus w_i \neq 0$ as $v_i \neq w_i$, thus by Axiom 1 the cost is one in each case. QED

Axiom 2 *checking if a binary path belongs to a non deterministic set costs* one *memory access.*

Theo 3 (exponential memory access) *If h is perfect, any* best *search algorithm for Formula (1) with Functions (2) may require up to 2^ℓ memory accesses.*

Proof From Theorem 1 and Lemma 1, we know that up to 2^ℓ valid paths may be enumerated.

Let us first assume that paths are enumerated in a deterministic order: at any point, a set of deterministic binary path B has been checked, and a new distinct binary path b' is considered. As $b' \notin B$, b' differs of any $b \in B$ by some ith bit. If $i < \ell$, from Lemma 8, x_i is near-independent of previous values, hence computing $f_{i+1}(x_i)$ or $g_{i+1}(x_i)$ will cost one memory access by Lemma 9. If $i = \ell$, then $x_{l-1} \oplus v_\ell$ and $x_{\ell-1} \oplus w_\ell$ are near-independent from Lemma 4 with $v = v_\ell \oplus w_\ell$, hence the t computation of one of these value will also cost one memory access by Axiom 1.

If the paths are not enumerated in a deterministic order, and as a *best* search cannot check the same path twice by Lemma 3, then it must check whether a binary path was already computed, adding one memory access by Axiom 2.

So for any *best* search, a valid path computation requires at least one memory access, thus computing the 2^ℓ valid paths costs at least 2^ℓ memory accesses. QED

Exponential memory fetches

Def 7 (memory fetch) *a unit of cost which involves accessing γ bits of memory.*

Def 8 (related evaluations) *function evaluations are* related *if they depend on the same value: for instance, $t(x)$, $t(x \oplus v)$ and $t(t(x))$ are all related through x.*

Axiom 3 *computing $t()$ values related through x, where t is a random function and x is near-independent of previously available values, costs* one *memory fetch.*

Lem 10 (fetched values) *If t is a random function in D, one memory fetch can bring up to $\frac{\gamma}{n}$ values related to t values to the processor.*

Proof This the number of distinct values held by a cache line. QED

Theo 4 (exponential memory fetches) *If h is perfect, any* best *search algorithm for Formula (1) with Functions (2) may require up to $\frac{n}{\gamma}2^{\ell}$ memory fetches.*

Proof From Theorem 3, we know that up to 2^{ℓ} memory accesses are performed by a *best* search. On the first memory access, a memory fetch is performed, bringing by Lemma 10 and if lucky $\frac{\gamma}{n}$ related and useful values from memory. Then another memory fetch is necessary for further values. QED

Spread and settings

Def 9 (coverage) *The coverage at level i for Formula (1), noted c_i, is the fraction of values x_i in D which belong to a valid path: $c_i = \frac{|\{x_i\}|}{2^n}$.*

Theo 5 (spread) *If for every $1 \leq i \leq \ell$, f_i, g_i are near-independent permutations in D, then for n large enough the coverage is: $c_0 = \frac{1}{2^n}$, $c_1 = \frac{2}{2^n}$ and $c_{i+1} \approx 2c_i - c_i^2$. Thus for the first iterations, the coverage grows exponentially: if $i < n$ then $c_i \approx 2^{i-n}$. It also converges quickly towards 1 as $1 - c_{i+1} \approx (1 - c_i)^2$.*

Proof There is one x_0, and two x_1 because the permutations are collision-free. Then by induction on i. Let coverage c_i be known. As the functions are permutations, all $2^n c_i$ distinct x_i values lead to $2^n c_i$ values through f_{i+1} and $2^n c_i$ values through g_{i+1}. Thanks to the near-independence, the collision probability is about c_i^2 and the formula is obvious. Fast initial divergence and convergence are simply derived from the iterative formula. QED

Theo 6 (our settings) *With $n = 22$ and $\ell = 26$, and assuming a perfect checksum, the average complexity of a best search is about half the worst-case given in the above theorems.*

Proof The worst-case is reached if there is only one solution to the server riddle and the client is unlucky enough to enumerate it the last.

On the first point, whether other solutions exist depends on collisions on the hash function: as it is assumed perfect, the probability of such an event only depends on the checksum size, and is about 2^{-m} at each trial. For our parameter $m = 32$ and $\ell = 26$, the probability of another solution is low (around $2^{-6} \approx 1.6\%$), thus does not impact significantly the average complexity.

On the second point, as the enumeration is without bias towards a solution, because the binary path is hidden by the hash function, and shamelessly neglecting the above collision probability, the solution is reached on average when half of the search space is explored.

Although the actual settings of t is pretty close to a random permutation because possibly built as such with a pseudo random-generator, our proposed software checksum is not perfect. Thus we do not claim that our full concrete proposal is necessarily as secure as demonstrated here under ideal conditions. QED