An (Almost) Constant-Effort Solution-Verification Proof-of-Work Protocol based on Merkle Trees
Technical Report A/390/CRI

Fabien Coelho
CRI, École des mines de Paris,
35, rue Saint-Honoré, 77305 Fontainebleau CEDEX, France.

November 24, 2007

Abstract

Proof-of-work schemes are economic measures to deter denial-of-service attacks: service requesters compute moderately hard functions that are easy to check by the provider. We present such a new scheme for solution-verification protocols. Although most schemes to date are probabilistic unbounded iterative processes with high variance of the requester effort, our Merkle tree scheme is deterministic, with an almost constant effort and null variance, and is computation-optimal.

1 Introduction

Economic measures to contain denial-of-service attacks such as spams were first suggested by Dwork and Naor [7]: a computation stamp is required to obtain a service. Proof-of-work schemes are dissymmetric: the computation must be moderately hard for the requester, but easy to check for the service provider. Applications include having uncheatable benchmarks [4], helping audit reported metering of web-sites [8], adding delays [19, 11], managing email addresses [9], making a data preservation protocol resistant to malign peers [20], or limiting abuses on peer-to-peer networks [10]. Proofs may be purchased in advance [1]. These schemes are formalized [12], and actual financial analysis is needed [15, 16] to evaluate their real impact. There are two flavors of protocol:

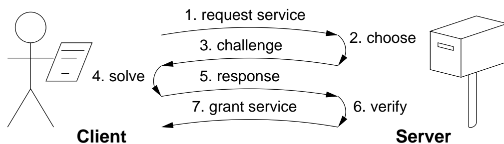

Figure 1: Challenge-Response protocol

Challenge-response protocols in Figure 1 assume an interaction between client and server, so that the service provider chooses the problem, say an item with some property from a finite set, and the requester must retrieve the item in the set. The solution is known to exist, the search time distribution is basically uniform, the solution is found on average when about half of the set has been processed, and standard deviation is about $\frac{1}{2\sqrt{3}} \approx 0.3$ of the mean.

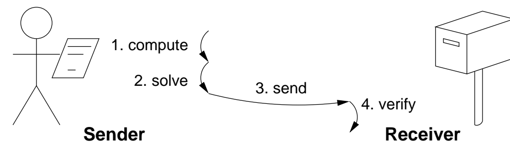

Figure 2: Solution-Verification protocol

Solution-verification protocols in Figure 2 do not assume such a link. The problem must be self-imposed, based somehow on the service description, say the intended recipient and date of a message. The target is usually a probabilistic property reached by iterations. The verification phase must check both the problem choice *and* the provided solution. Such iterative searches have a constant probability of success at each trial, resulting in a shifted geometrical distribution, the mean is the inverse of the success probability, and the standard deviation nearly equals the mean. The resulting distribution has a long tail, as the number of iterations to success is not bounded: about every 50 searches an unlucky case requires more than 4 times the average to complete.

We present a new proof-of-work solution-verification scheme based on Merkle trees with an almost constant effort and null variance for the client. The solution costs about $2N$, $P \cdot \ln(N)$ data is sent, and the verification costs $P \cdot \ln(N)$, with $P = 8 \log_2(N)$ a good choice. This contribution is theoretic with a low variance requester effort, but also practical as the computation-optimal function has an interesting work-ratio.

Section 2 discusses proof-of-work schemes suggested to date, and analyzes their optimality and the computation distribution of solution-verification variants. Section 3 describes our new computation-optimal scheme based on Merkle trees built on top of the service description. The solution involves the computation of most of the tree, although only part of it is sent thanks to a feedback mechanism that selects only some leaves. Section 4 computes a cost lower bound to find a proof, then outlines two beneficial attacks.

2 Related work

Let the *effort* be the amount of computation of the requester as a function of the provider work, and the *work-ratio* the effort divided by the provider work. We introduce two optimality criteria to analyze proof-of-work schemes, then discuss solution-verification protocols suggested to date with respect to these criteria and to the work distribution on the requester side. Challenge-response only functions [18, 13, 22] are not discussed further in this section.

Proof-of-work schemes may be: (a) *communication-optimal* if the amount of data sent on top of the service description D is minimal. For solution-verification iterative schemes it is ln(work-ratio) to identify the found solution. For challenge-response protocols, it would be ln(search space size). This criterion emphasizes minimum impact on communications, and is met by iterative schemes that return an index. (b) *computation-optimal* if the challenge or verification work is simply linear in the amount of communicated data, which it must at least be if the data are processed. This criterion mitigates denial-of-service attacks on service providers, as it avoids fake proof-of-works that would require significant resources to disprove. A scheme meeting both criteria is deemed optimal.

Three proof-of-work schemes are suggested by Dwork and Naor [7]. One is a formula (integer square root modulo a large prime $p \equiv 3 \bmod 4$), as computing a square root is more expensive than squaring the result to check it. Assuming a naïve implementation, it costs $\ln(p)^3$ to compute, $\ln(p)$ to communicate and $\ln(p)^2$ to check. The search effort is constant and the variance is null, but the effort is not very interesting, and is not optimal in any sense. Better implementations reduce both solution and verification complexities. If $p \equiv 1 \bmod 4$, the square root computation with the Tonelli-Shanks algorithm involves a non deterministic step with a geometrical distribution. The next two schemes present shortcuts which allow some participants to generate cheaper stamps. They rely on forging a signature without actually breaking a private key. One uses the Fiat-Shamir signature with a weak hash function for which an inversion is sought by iterating, with a geometrical distribution of the effort. The computation costs $E \cdot \ln(N)^2$, the communication $\ln(N)$ and the verification $\ln(N)^2$, where $N \gg 2^{512}$ is needed for the scheme security and the arbitrary effort E is necessarily much smaller than N, thus the scheme is not optimal. The other is the Ong-Schnorr-Shamir signature broken by Pollard, with a similar non-optimality and a geometrical distribution because of an iterative step.

Some schemes [3, 8, 21] seek *partial* hash inversions. Hashcash [3] iterates a hash function on a string involving the service description and a counter, and is optimal. This string was computed in 400 seconds on a 2005 laptop:

```
1:28:170319:hobbes@comics::7b7b973c8bdb0cb1:147b744d
```

It allows to send an email to *hobbes* on March 19, 2017. The last part is the hexadecimal counter, and the SHA1 hash of the whole string begins with 28 binary zeros. Franklin and Malkhi [8] builds a hash sequence: it statistically catches cheaters, but the verification may be expensive. Wang and Reiter [21] allows the requester to tune the effort to improve its priority.

Memory-bound schemes [2, 6, 5] seek to reduce the impact of the computer hardware performance on computation times. All solution-verification variants are based on an iterative search targeting a partial hash inversion, thus have a geometrical distribution of success, and are communication-optimal. However only the last of these memory-bound solution-verification schemes is computation-optimal.

3 Scheme

This sections describes our (almost) constant-effort and null variance solution-verification proof-of-work scheme. The client is expected to compute a Merkle tree, but is required to give only part of the tree for verification by the service provider. A feedback mechanism uses the root hash so that the given part cannot be known in advance, thus induces the client to compute most of the tree for a solution. Finally choice of parameters and a memory-computation implementation trade-off are discussed.

3.1 Merkle tree

Let h be a cryptographic hash function from anything to a domain of size 2^m. The complexity of such functions usually depends linearly in the input length by step, and for our purpose the input is short thus, computations only involve one step. Let D be a service description, for instance an ascii string as `hobbes@comics:20170319:0001`, and $s = h(D)$ its hash. The Merkle binary hash tree [17] of depth d ($N = 2^d$) is computed as: (1) leaf digests $n_{N-1+i} = h(s\|i)$ for i in $0 \ldots N-1$; (2) inner nodes are propagated upwards $n_i = h(s\|n_{2i+1}\|n_{2i+2})$ for i in $N-2 \ldots 0$. Root hash n_0 is computed with $2N$ calls to h, half of which for leaf computations, one for the service, and the remainder for the internal nodes.

3.2 Feedback

Merkle trees help manage Lamport signatures [14]: a partial tree allows to check that some leaves belong to the full tree. We use this property to generate our proof of work: the requester returns such a partial tree to show that selected leaves belong to the tree and were indeed computed. However, what particular leaves are needed must not be known in advance, otherwise it would be easy to generate a partial tree just with those

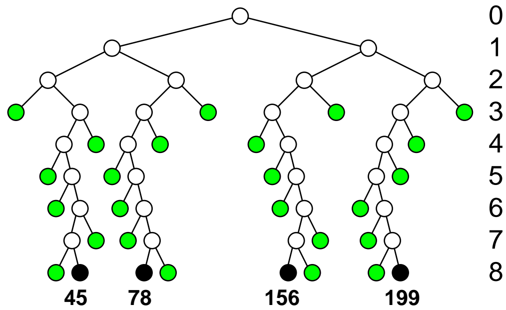

Figure 3: Merkle tree proof ($P = 4$, $N = 2^8$)

leaves. Returned leaves are selected based on the root hash, so as to depend on the tree computation.

This feedback phase chooses P evenly-distributed independent leaves derived from the root hash as partial proofs of the whole computation. A cryptographic approximation of such an *independent-dependent* derivation is to seed a pseudo-random number generator from the root hash and to extract P numbers corresponding to leaves in P consecutive segments of size $\frac{N}{P}$. These leaf numbers and the additional nodes necessary to check for the full tree constitute the proof-of-work. Figure 3 illustrates the data sent for 4 leaf-proofs (black) and the intermediate hashes that must be provided (green) or computed (white) on a 256-leaves tree.

3.3 Verification

The service provider receives the required service description D, P leaf numbers, and the intermediate hashes necessary to compute the root of the Merkle tree which amount to about $P \cdot \log_2(N) \cdot m$ bits.

The server checks the consistency of the partial tree by recomputing the hashes starting from service hash s and leaf numbers and up to the root hash by using the provided intermediate node hashes, and then by checking that this root hash leads to the provided choice of leaves. This requires about $P \cdot \log_2(N)$ hash computations for the tree, and some computations of the pseudo-random number generator. This phase is computation-optimal as each data is processed a fixed number of times by the hash function for the tree and generator computations.

The root hash is not needed to validate the Merkle tree: it is computed anyway by the verification, and if enough leaves are required its value is validated indirectly when checking that the leaves are indeed the one derived from the root hash.

3.4 Choice of parameters

Let us discuss the random generator, the hash function, the tree depth, and the number of proofs.

The pseudo-random number generator supplies $P \cdot \log_2(\frac{N}{P})$ bits (14 bits per proof for $N = 2^{22}$ and $P = 256$) to choose the evenly-distributed leaves. Standard generators can be seeded with the root hash. To add to the cost of an attack without impact on the verification complexity, the generator may rely on h: compute $r = h^P(s \| n_0)$, and choose the ith leaf as $l_i = h(r \| i) \bmod \frac{N}{P}$, so that about P hash computations are needed to test a partial tree, as discussed in the Section 4.1.

The hash width may be different for the description, lower tree, upper tree, and generator. The description hash must avoid collisions which would lead to reusable trees; the generator hash should keep as much entropy as possible; in the upper part of the tree, a convenient root hash should not be targettable, and the number of distinct root hashes should be large enough so that it is not worth precomputing them, as well as to provide a better initial entropy. A standard cryptographic hash as SHA1 is advisable in these cases. For the lower tree and leaves, the smaller m the better, as it drives the amount of temporary data and the proof size. Tabulating node hashes for reuse is not interesting as they depend on s and if $2^{2m} \gg 2N$. Moreover it should not be easily invertible, so that a convenient hash cannot be targeted by a search process at any point. A sufficient condition is $2^m > 2N$: one inversion costs more than the whole computation. For our purpose, the lower tree hash may be folded to $m = 24$ for $N = 2^{22}$.

The Merkle tree depth leads to the number of leaves N and the expected number of hash computations $2N$. The resource consumption required before the service is provided must depend on the cost of the service. For emails, a few seconds per recipient seems reasonable. With SHA1, a depth of 22 leads to 2^{23} hash calls and warrants this effort on my 2005 laptop. For other hash functions, the right depth depends on the performance of these functions on the target hardware.

The smaller the number of proofs, the better for the communication and verification involved, but if very few proofs are required a partial computation of the Merkle tree could be greatly beneficial. We choose $P = 8 \log_2(N)$, maybe rounded up to a power of two to ease the even distribution. Section 4.2 shows this as enough to make the service requester compute most of the tree. With this number of proofs, the solution effort is $e^{\frac{w}{2}}$. It is not communication-optimal: proofs are a little bit large, about 11 KB for $N = 2^{22}$.

3.5 Memory-computation trade-off

The full Merkle tree needs about $2N \cdot m$ bits if it is kept in memory to extract the feedback. A simple yet efficient trade-off is to keep only the upper part of the tree, dividing the memory requirement by 2^t, at the price of $P \cdot 2^{t+1}$ hash computations to rebuild the subtrees that contain the proofs. The limit case recomputes the full tree once the needed leaves are known.

4 Attacks

In the above protocol, the requester uses $2N$ hash computations for the Merkle tree, but the provider only $P \cdot \log_2(N) \approx 8\log_2(N)^2$ to verify the extracted partial tree, and both side must run the generator. This section discusses attacks which reduce the requester work by computing only a fraction of the tree and being lucky with the feedback so that required leaves are available. We first compute a lower bound for the cost of finding a solution depending on the parameters, then we discuss two attacks.

4.1 Partial tree

In order to cheat one must provide a matching tree, *i.e.*: (a) a valid partial tree starting from the service hashes, *or the tree itself is rejected* ; (b) with valid leaves choice based on the root hash, *or the feedback fails.* As this tree is built from a non-invertible hash function, the hash computations must have been performed by the requester at least for the accepted partial tree, or the verification would very likely fail.

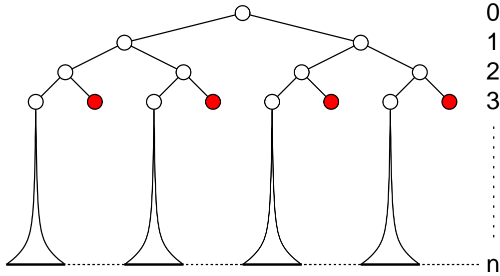

Figure 4: Partial Merkle tree ($f = 0.5$, $P = 4$)

Let us assume that the attacker builds a partial tree involving a fraction f of the leaves, where missing hash values are filled-in randomly, as outlined in Figure 4: evenly-distributed proofs result in 4 real hashes at depth 2, computed from 4 fake hashes introduced at depth 3 to hide the non-computed subtrees, and 4 real hashes coming from the subtrees. At the leaf level, half of the hashes are really computed.

Once the root hash is available, the feedback leaves can be derived. If they are among available ones, a solution has been found and can be returned. The probability of this event is f^P. It is quickly reduced by smaller fractions and larger numbers of proofs. If the needed proof leaves are not all available, no solution was found. From this point, the attacker can either start all over again, or reuse only part of the tree at another attempt, or alter the current tree. The later is the best choice. This tree alteration can either consist of changing a fake node (iteration at constant f), or of adding new leaves (extending f).

We are interested in the expected average cost of the search till a suitable root hash which points to available leaves is found. Many strategies are possible as iterations or extensions involving any subset of leaves can be performed in any order. However, each trial requires the actual root hash for a partial tree and running the generator. Doing so adds to the current total cost of the tree and to the cost of later trials.

4.2 Cost lower bound

A conservative lower bound cost for a successful attack can be computed by assuming that every added leaf is tried up to the root hash with no over-cost for the queue to reach the root nor for running the generator. We first evaluate an upper bound of the probability of success for these partial trees, which is then used to derive a lower bound for the total cost.

If we neglect the even distribution of proofs, the probability of success at iteration i (an ith leaf is added in the tree) is $\rho_i = (\frac{i}{N})^P$, and the probability of getting there is $(1-\sigma_{i-1})$ where σ_i is the cumulated probability of success up to i: $\sigma_0 = 0$, $\sigma_i = \sigma_{i-1} + (1-\sigma_{i-1})\rho_i$, and $\sigma_N = 1$, as the last iteration solves the problem with $\rho_N = 1$. The $(1-\sigma_{i-1})\rho_i$ term is the global probability of success at i: the computation got there (the problem was not solved before) and is solved at this very iteration. As it is lower than ρ_i:

$$\sigma_j \leq \sum_{i=0}^{j} \rho_i \leq \int_0^{\frac{j+1}{N}} N x^P dx = \frac{N}{P+1}(\frac{j+1}{N})^{P+1} \quad (1)$$

If $c(i)$ is the increasing minimal cost of building a tree with i leaves, the average cost $\mathcal{C}$ for the requester is:

$$\begin{aligned}\mathcal{C}(N,P) &\geq \sum_{i=1}^{N} c(i)(1-\sigma_{i-1})\rho_i = \sum_{i=1}^{N} c(i)(\sigma_i - \sigma_{i-1}) \\ &= \sum_{i=1}^{\ell-1} c(i)(\sigma_i - \sigma_{i-1}) + \sum_{i=\ell}^{N} c(i)(\sigma_i - \sigma_{i-1}) \\ &\geq 0 + c(\ell)(\sigma_N - \sigma_{\ell-1}) \\ &= c(\ell)(1-\sigma_{\ell-1}) \geq c(\ell)(1-\sigma_\ell)\end{aligned}$$

The cost is bounded by cutting the summation at ℓ chosen as $\frac{\ell+1}{N} = (\frac{1}{N})^{\frac{1}{P+1}}$. The contributions below this limit are zeroed, and those over are minimized as $c(\ell) \geq 2\ell$ and $(1-\sigma_\ell)$ is bound with Equation (1) so that $(1-\sigma_\ell) \geq (1-\frac{1}{P+1}) = \frac{P}{P+1}$ hence:

$$\mathcal{C}(N,P) \geq (\frac{1}{N})^{\frac{1}{P+1}} \frac{P}{P+1}(2N) \quad (2)$$

Figure 5 plots this estimation. A corner is empty where the number of proofs is greater than the number of leaves. Choosing $P = 8\log_2(N)$ and assuming $N \geq 2^7$:

$$\mathcal{C}(N) \geq (\frac{1}{2})^{\frac{1}{8}} \frac{8\log_2(N)}{8\log_2(N)+1}(2N) \geq 0.9(2N)$$

Whatever the attack strategy, for our suggested number of proofs and a tree of depth 7 or more, a requester will have to compute at least 90% of the full Merkle tree on average.

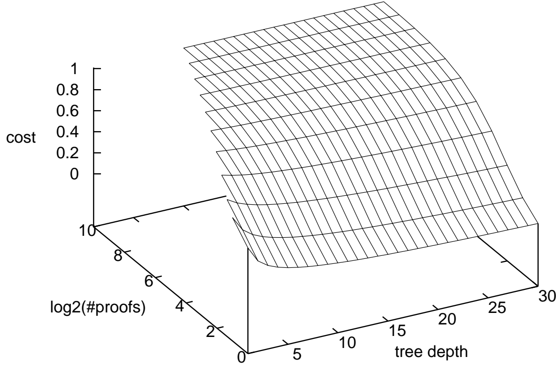

Figure 5: Relative cost lower bound – Equation (2)

4.3 Iterative attack

Let us investigate a strategy that fills a fraction of the tree with fake hashes introduced to hide non computed leaves, and then iterates by modifying a fake hash till success, without increasing the number of leaves. The resulting average cost is shown in Equation (3). The first term approximates the hash tree computation cost for the non-faked leaves and nodes, and is a minimum cost for the attack with a given fraction f. The second term is the average iteration cost for a solution, by trying faked hash values from depth $\log_2(P)+1$ thanks to the even-distribution, and another P for the generator.

$$\mathcal{C}_{\text{iter}}(f) \approx 2Nf + (P + \log_2(P) + 1)\frac{1}{f^P} \tag{3}$$

If f is small, the second term dominates, and the cost is exponential. If f is close to 1, the first linear term is more important and the cost is close to the full tree computation. This effect is illustrated in Figure 6 for different number of proofs: few proofs lead to very beneficial fractions: many proofs make the minimum of the functions close to the full tree computation.

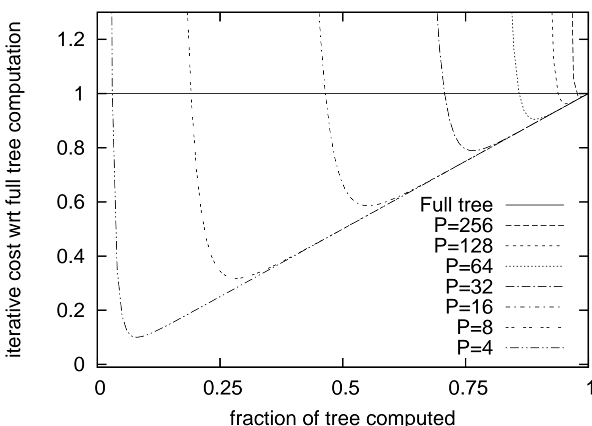

Figure 6: Iterative cost for fraction f with $N = 2^{22}$

$$\mathcal{F}(N, P) = \sqrt[P+1]{\frac{P(P + \log_2(P) + 1)}{2N}} \tag{4}$$

Equation (4), the zero of (3) derivative, gives the best fraction of this iterative strategy for a given size and number of proofs. $\mathcal{F}(2^{22}, 256) = 0.981$ and the cost is 0.989 of the full tree, to be compared to the 0.9 lower bound computed in the previous subsection. Whether a significantly better strategy can be devised is unclear. A numerical cost lower bound gives a 0.961 multiplier for the same parameters. In order to reduce the effectiveness of this attack further, the hash-based generator may cost up to $P \cdot \log_2(N)$ to derive r without impact on the overall verification complexity, but at the price of doubling the verification cost.

This successful attack justifies the *almost* constant-effort claim: either a full tree is computed and a solution is found with a null variance, or some partial-tree unbounded attack is carried out, maybe with a low variance, and costing at least 90% of the full tree.

4.4 Skewed feedback attack

Let us illustrate the impact of a non-independent proof selection by the pseudo-random number generator. We assume an extreme case where the generator directly uses the first bits of the root hash as a unique leaf index in all $\frac{N}{P}$ chunks: the selected leaves would be $\{k, k + \frac{N}{P}, k + 2\frac{N}{P}, \ldots\}$. Then in the partial tree attack the requester could insure that any leaf k computed in the first chunk have their corresponding shifted leaves in the other chunks available. Thus, when hitting one leaf in the first chunk, all other leaves follow, and the probability of a successful feedback is f instead of f^P. $N = 2^{22}$ and $P = 256$ lead to $0.002(2N)$, a 474 speedup of the attack efficiency.

5 Conclusion

Proof-of-work schemes deter denial-of-service attacks by requiring costly computations from the requester that are easy to verify by the provider. As solution-verification protocol variants do not assume any interaction between requesters and providers, the computations must be self-imposed, based somehow on the expected service. Most of these schemes are unbounded iterative probabilistic searches with a high variance of the requester effort. We have made the following contributions about proof-of-work schemes:

1. two optimality criteria: communication-optimal if the minimum amount of data is sent; computation-optimal if the verification is linear in the sent data;
2. a computation-optimal proof-of-work solution-verification scheme based on Merkle trees with a $e^{\frac{w}{2}}$ effort, for which the work on the requester side is bounded and the variance is null;

3. a conservative lower bound of the cost of finding a solution at 90% of the full computation, which shows that our chosen number of proofs is sound;

4. a successful attack with a small 1% gain for our parameters, which involves a large constant cost and a small iterative unbounded part, thus results in a low overall variance.

These contributions are both theoretic and practical. Our solution-verification scheme has a bounded, constant-effort solution. In contrast to iterative probabilistic searches for which the found solution is exactly checked, but the requester's effort is probably known with a high variance, we rather have a probabilistic check of the proof-of-work, but the actual solution work is quite well known with a small variance thanks to the cost lower bound. Moreover our scheme is practical, as it is computation-optimal thus not prone to denial-of-service attacks, and, although not optimal, the communication induces an interesting work-ratio. The only other bounded solution-verification scheme is a formula with a $w^{1.5}$ effort, which is neither communication nor computation-optimal. Whether a bounded fully optimal solution-verification scheme may be built is an open question.

One may finally reflect that wasting CPU-cycles in heat for pointless computations is not ecological. However, denial-of-service attacks are not green either, and deterring them may avoid wasting significant resources on the provider side.

Thanks

Françis Maisonneuve, Vincent Bachelot and Alexandre Aillos, helped to find a lower bound for the cost of computing a matching partial tree; Pierre Jouvelot and Ronan Keryell helped to improve the redaction.

References

[1] M. Abadi, A. Birrell, B. Mike, F. Dabek, and T. Wobber. Bankable postage for network services. In *8th Asian Computing Science Conference, Mumbai, India*, volume 2986 of *LNCS*, pages 72–90. Springer-Verlag, Dec. 2003.

[2] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. In *10th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, February 2003.

[3] A. Back. Hashcash package announce. `http://www.hashcash.org/papers/announce.txt`, Mar. 1997.

[4] J.-Y. Cai, R. R. Lipton, R. Sedgewick, and A. C.-C. Yao. Towards uncheatable benchmarks. In *Eighth IEEE Annual Structure in Complexity Conference*, pages 2–11, San Diego, California, May 1993.

[5] F. Coelho. Exponential memory-bound functions for proof of work protocols. Research Report A-370, CRI, École des mines de Paris, Sept. 2005. Also Cryptology ePrint Archive, Report 2005/356.

[6] C. Dwork, A. Goldberg, and M. 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.

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

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

[9] E. Gabber, M. Jakobsson, Y. Matias, and A. J. Mayer. Curbing junk e-mail via secure classification. In *Financial Cryptography*, pages 198–213, 1998.

[10] F. Garcia and J.-H. Hoepman. Off-line Karma: A Decentralized Currency for Peer-to-peer and Grid Applications. In *Applied Cryptography and Network Security – ACNS*, number 3531 in LNCS, pages 364–377, June 2005. 3rd Internationnal Conference.

[11] D. M. Goldschlag and S. G. Stubblebine. Publicly verifiable lotteries: Applications of delaying functions. In R. Hirschfeld, editor, *Financial Cryptography'98*, 1998.

[12] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In *Comms and Multimedia Security 99*, 1999.

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

[14] L. Lamport. Constructing digital signatures from a one-way function. Technical Report SRI-CSL-98, SRI International Computer Science Laboratory, Oct. 1979.

[15] B. Laurie and R. Clayton. "proof-of-work" proves not to work. In *WEAS 04*, May 2004.

[16] D. Liu and L. J. Camp. Proof of Work can Work. In *Fifth Workshop on the Economics of Information Security*, June 2006.

[17] R. C. Merkle. *Secrecy, Authentification, and Public Key Systems*. PhD thesis, Stanford University, Dpt of Electrical Engineering, June 1979.

[18] R. L. Rivers, A. Shamir, and D. Wagner. Time-lock puzzles and timed-release crypto. Technical Report MIT/LCS 684, MIT, 1996.

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

[20] D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, and M. 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] X. Wang and M. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *IEEE Symposium on Security and Privacy 03*, May 2003.

[22] B. Waters, A. Juels, J. A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for DoS resistance. In *11th ACM Conference on Computer and Communications Security*, 2004.

Typeset with LaTeX, revision 742.