An (Almost) Constant-Effort Solution-Verification Proof-of-Work Protocol based on Merkle Trees (extended and colored version of [7])

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

Abstract

Proof-of-work schemes are economic measures to deter denial-of-service attacks: service requesters compute moderately hard functions the results of which 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 [9]: 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 [5], helping audit reported metering of web sites [10], adding delays [21, 13], managing email addresses [11], limiting abuses on peer-to-peer networks [22, 12] or streaming protocols [1]. Proofs may be purchased in advance [2]. These schemes are formalized [14], and actual financial analysis is needed [17, 18] to evaluate their real impact. There are two protocol flavors:

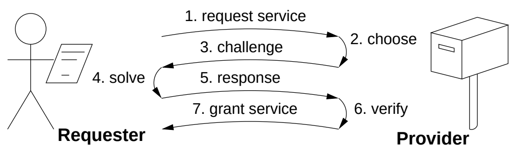

Figure 1: Challenge-Response Protocol

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.

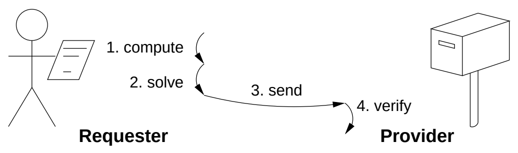

Figure 2: Solution-Verification Protocol

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 perhaps 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 number of iterations to complete (the probability of not succeeding in 4 times the average is about $e^{\frac{-4a}{a}} = e^{-4} \approx \frac{1}{50}$).

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. When considering a Merkle tree with N leaves, the solution costs about $2N$, $P \cdot \ln(N)$ data is sent, and the verification costs $P \cdot \ln(N)$ with $P = 8 \cdot \ln_2(N)$ a good choice. This contribution is theoretical with a constant requester effort, which is thus bounded or of possibly low variance, but also practical as our scheme is computation-optimal and 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 scheme based on Merkle trees built on top of the service description. This scheme is shown computation-optimal, but is not communication-optimal. The solution involves the computation of most of the tree, although only part of it is sent thanks to a feedback mechanism which selects only a subset of the leaves. Section 4 computes a cost lower bound for a proof, then outlines two attacks beneficial to the service requester. The effort of our scheme is constant, thus bounded and with a null variance. However we show an iterative attack, which is not upper-bounded, and which results in a small gain. Together with the demonstrated lower-bound cost of a proof, it justifies our *almost* claim.

2 Related 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 [20, 15, 24] are not discussed further here.

Let the *effort* $E(w)$ be the amount of computation of the requester as a function of provider work w, and the *work-ratio* the effort divided by the provider work. 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 about ln(work-ratio) to identify the found solution: the work-ratio is the number of iterations performed over a counter to find a solution, and it is enough to just return the value of this counter for the provider to check the requester proof. For challenge-response protocols, it would be ln(search space size). This criterion emphasizes minimum impact on communications. (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 is processed (*i.e.* simply received, and possibly checked). This criterion mitigates denial-of-service attacks on service providers, as fake proof-of-works could require significant resources to disprove. A scheme meeting both criteria is deemed optimal.

Three proof-of-work schemes are suggested by Dwork and Naor [9]. 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 cost is deterministic, but the $w^{1.5}$ effort is not very interesting, and is not optimal. 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 iteration, 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.

ref	*effort*	*var*	*comm.*	*work*	*constraints*
[9]1	$\ln(p)^3$	0	$\ln(p)$	$\ln(p)^2$	p large prime
[9]2	$E\ln(N)^2$	> 0	$\ln(N)$	$\ln(N)^2$	$N \gg 2^{512}$
[4]	E	=	$\ln(E)$	$\ln(E)$	
[3]	$E\ell$	=	$\ln(E)$	ℓ	typical $\ell = 2^{13}$
[8]	$E\ell$	=	$\ln(E)$	ℓ	$E \ll 2^\ell, \ell > 2^{10}$
[6]	E	=	$\ln(E)$	$\ln(E)$	
here	$2N$	0	$P\ln(N)$	$P\ln(N)$	$P = 8 \cdot \ln_2(N)$

Table 1: Comparison of Solution-Verification POW

Some schemes [4, 10, 23] seek *partial* hash inversions. Hashcash [4] iterates a hash function on a string involving the service description and a counter, and is optimal. The following stamp computed in 400 seconds on a 2005 laptop:

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

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 [10] build a hash sequence that statistically catches cheaters, but the verification may be expensive. Wang and Reiter [23] allow the requester to tune the effort to improve its priority.

Memory-bound schemes [3, 8, 6] seek to reduce the impact of the computer hardware performance on computation times. All solution-verification variants are based on an iterative search which target a partial hash inversion, and 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.

Table 1 compares the requester cost and variance, communication cost, and provider checking cost, of solution-verification proof-of-work schemes, with the notations used in the papers.

3 Scheme

This section describes our (almost) constant-effort and null variance solution-verification proof-of-work scheme. The client is expected to compute a Merkle tree which depends on a service description, 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. The notations used thoroughly in this paper are summarized in Table 2. The whole scheme is outlined in Figure 8.

Symbol	*Definition*
w	provider checking work
$E(w)$	requester effort
D	service description, a string
h	cryptographic hash function
m	hash function bit width
s	service hash is $h(D)$
h_s	service-dependent hash function
d	depth of Merkle binary hash tree
N	number of leaves in tree is 2^d
P	number of proofs expected
n_i	a node hash in the binary tree
n_0	root hash of the tree
r	leaf selector seed is $h_s^P(n_0)$

Table 2: Summary of notations

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 is usually stepwise linear in the input length. For our purpose the input is short, thus computations only involve one step. Let D be a service description, for instance a string such as `hobbes@comics:20170319:0001`. Let $s = h(D)$ be its hash. Let $h_s(x) = h(x\|s)$ be a service-dependent hash. The Merkle binary hash tree [19] of depth d ($N = 2^d$) is computed as follows: (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 for leaf computations, one for service s, and the remainder for the internal nodes of the tree. The whole tree *depends* on the service description as s is used at every stage: reusing such a tree would require a collision of service description hashes. Figure 3 illustrates the Merkle tree construction from leaves to root. Note that the hash function may be dithered by adding an identifier of the computed node to the hashed value: $n_i = h_s(i\|n_{2i+1}\|n_{2i+2})$.

3.2 Feedback

Merkle trees help manage Lamport signatures [16]: a *partial* tree allows to check quickly that some leaves

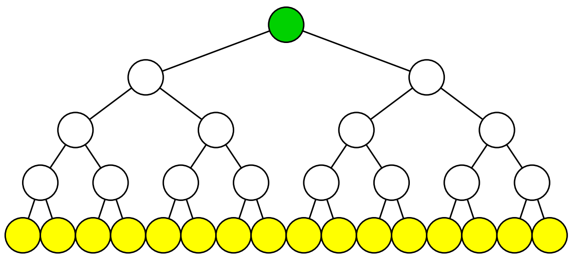
Figure 3: Merkle tree

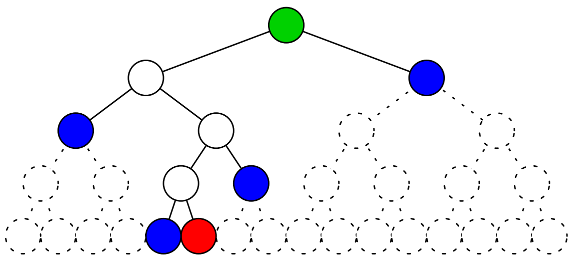
Figure 4: Merkle tree proof

belong to the full tree by checking that they actually lead to the root hash. This is outlined in Figure 4, where 4 intermediate hashes are enough to show that belong to the tree by recomputing the root hash.

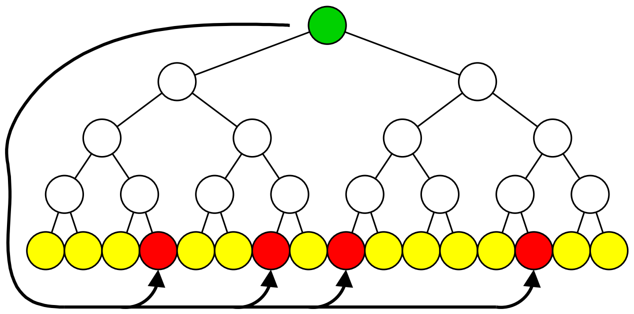
Figure 5: Merkle tree feedback

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 thus 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 leaves and to provide random values for the other branches. Thus we select returned leaves based on the root hash, so that they depend on the whole tree computation. Finally, in order for the scheme to be interesting on the provider size, the number of selected leaves must be quite small, as the partial computation starting from these leaves will have to be performed for verification.

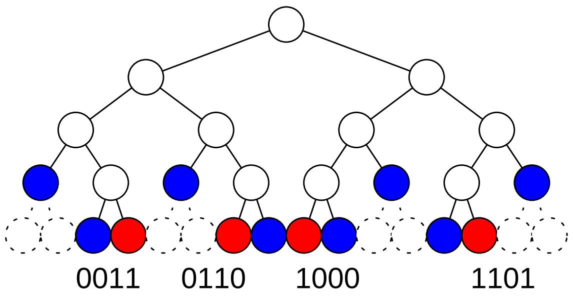

Figure 6: Proof-of-work Communication

The feedback phase chooses P evenly-distributed independent leaves derived from the root hash as partial proofs of the whole computation, as suggested by Figure 5 A cryptographic approximation of such an *independent-dependent* derivation is to seed a pseudo-random number generator from root hash n_0 and to extract P numbers corresponding to leaves in P consecutive chunks 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 6 illustrates the data sent for 4 leaf-proofs (red) and the intermediate hashes that must be provided (blue) or computed (white) on a 256-leaf tree. They are evenly distributed as one leaf is selected in every quarter of the tree, so balanced branches only meet near the root.

3.3 Verification

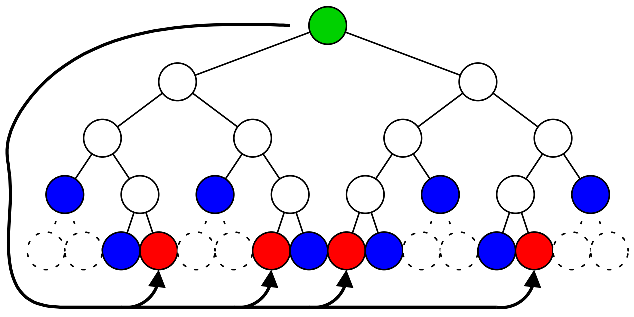

Figure 7: Proof-of-work verification

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 $P \cdot \ln_2(\frac{N}{P}) \cdot (m+1)$ bits: $P \cdot \ln_2(\frac{N}{P})$ for the leaf numbers inside the chunks, and $P \cdot \ln_2(\frac{N}{P}) \cdot m$ for the intermediate hashes.

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 using the provided intermediate node hashes, and then by checking the feedback choice, *i.e.* that the root hash does lead to the provided leaves, as outlined in Figure 7. This requires about $P \cdot \ln_2(N)$ 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.

Note that the actual root hash is not really 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 seeded generator.

3.4 Choice of parameters

Let us discuss the random generator, the hash function h and its width m, the tree depth d ($N = 2^d$) and the number of proofs P.

The pseudo-random number generator supplies $P \cdot \ln_2(\frac{N}{P})$ bits ($14 = 22 - 8$ bits per proof for $N = 2^{22}$ and

```
**solution work** by requester
  define service description:
    $D =$ hobbes@comics:20170319:0001
  compute service description hash:
    $s = h(D) =$ 36639b2165bcd7c724...
  compute leaf hashes:
    for $i$ in $0 \dots N-1$: $n_{N-1+i} = h_s(i)$
  compute internal node hashes:
    for $i$ in $N-2 \dots 0$: $n_i = h_s(n_{2i+1} \| n_{2i+2})$
  compute generator seed
    $r = h_s^P(n_0)$
  derive leaf numbers in each $P$ chunk
    for $j$ in $0 \dots P-1$: $\ell_j = \mathcal{G}(r, j)$

**communication** from requester to provider
  send service description $D$
  send $P$ leaf numbers $\ell_j$ for $j \in (0 \dots P-1)$
  for each paths of selected leaves
    send intermediate lower tree node hashes
  that's $P \ln_2(\frac{N}{P})$ hashes of width $m$

**verification work** by provider
  check service description $D$
    *do I want to provide this service?*
  compute service hash
    $s = h(D)$
  compute root hash $n_0$
    from $\ell_j$ and provided node hashes
  compute generator seed
    $r = h_s^P(n_0)$
  derive leaf numbers in each $P$ chunk
    for $j$ in $0 \dots P-1$: $\ell'_j = \mathcal{G}(r, j)$
  check whether these leaf numbers were provided
    $\forall j \in (0 \dots P-1)$, $\ell_j = \ell'_j$
```

Figure 8: Scheme Outline

$P = 256 = 2^8$) to choose the evenly-distributed leaves. Standard generators can be seeded directly with the root hash. To add to the cost of an attack without impact on the verification complexity, the generator seed may rely further on h by using seed $r = h_s^P(n_0)$ (h_s composed P times over itself), so that about P hash computations are needed to test a partial tree, as discussed in Section 4.1. The generator itself may also use h, say with the j-th leaf in the j-th chunk chosen as $\ell_j = \mathcal{G}(r, j) = h_s(j \| r) \bmod \frac{N}{P}$ for j in $0 \dots P-1$.

The hash width may be different for the description, lower tree (close to the leaves), upper tree (close to the root), and generator. The description hash must avoid collisions which would lead to reusable trees; the generator hash should keep as much entropy as possible, especially if the root hash is iterated P times to compute the seed; in the upper part of the tree, a convenient root hash should not be targetable, 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 typical hash inversion attack in the upper tree would be to chose an arbitrary root hash, to derive the generator seed and selected leaf numbers, and then to build a partial hash tree only with those leaves as discussed in Section 4, and finally to target the chosen root hash by iterating. Thus a strong cryptographic hash 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 because they all 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 hash inversion costs more than the *whole* computation. For our purpose, with $N = 2^{22}$, the lower tree hash may be folded to $m = 24$. The impact of choosing $m = \ln_2(N) + 2$ is not taken into account in our complexity analyses because h is assumed a constant cost for any practical tree depth: it would not change our optimality result to do so, but it would change the effort function to $e^{\sqrt[3]{w}}$.

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 should depend on the cost of the service. For emails, a few seconds computation per recipient seems reasonable. With SHA1, depth $d = 22$ 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 number of leaves also induces the number of required proofs, hence the total proof size, as discussed hereafter.

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 to the requester. We choose $P = 8 \cdot \ln_2(N)$, maybe rounded up to a power of two to ease the even distribution. This is a small number of leaves for most pratical settings: a depth 32 tree involving $2^{32} = 4$ billion leaves would require only 256 proofs. Section 4.2 shows that this value induces the service requester to compute most of the tree. With this number of proofs, the solution effort is $e^{\sqrt{w}}$ (verification work $w = \mathcal{O}(\ln(N)^2)$, and provider effort is $2N \approx e^{\sqrt{w}}$).

It is not communication-optimal: proofs are a little bit large, for instance with SHA1 as a hash and with $N = 2^{22}$ it is about 11 KB (that is $256 \cdot (22 - 8) \cdot (24 + 1)$ bits), although around 22 bits are sufficient for a counter-based technique. The communication volume can be reduced by selecting a slower hash function, however a corresponding increased price will be paid by the service provider: For a 64-times slower hash function and the same solution time for the requester, the tree depth is reduced to $d = 16$, number of proofs $P = 128$, communication is 2.7 KB (a factor 4 reduction), and verification costs is reduced by factor 2.7 hash operations, but the verification time is increased by a factor 24 as these operations are more costly. From the service provider perspective, the faster the hash the better at an equal cost for the service requester, but also the larger the required communication.

3.5 Memory-computation trade-off

The full Merkle tree needs about $2N \cdot m$ bits if it is kept in memory, to be able to extract the feedback hashes once the required leaves are known. A simple 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 needs only $P \cdot \ln_2(N) = 8 \cdot (\ln_2(N))^2$ 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 partial 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 cryptographic hash function, the successful attacker must have computed the provided partial Merkle tree root hash and its leaf derivations: otherwise the probability of returning a matching partial tree by chance is the same as finding a hash inversion.

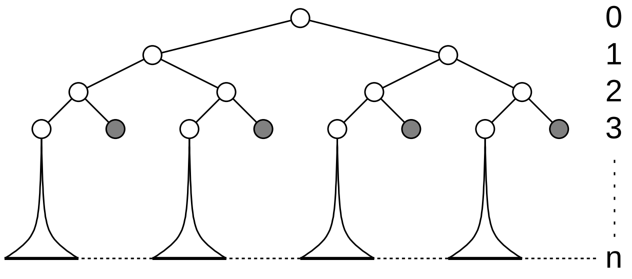

Figure 9: Partial Merkle tree ($f = 0.5$, $P = 4$)

Let us assume that the attacker builds a partial tree involving a fraction f of the leaves ($0 \leq f \leq 1$), where missing hash values are filled-in randomly, as outlined in Figure 9: evenly-distributed proofs result in 4 real hashes at depth 2, computed from 4 fake hashes (in grey) introduced at depth 3 to hide the non-computed subtrees, and 4 real hashes coming from the real subtrees. Half leaf 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, reuse only part of the tree at another attempt, or alter the current tree. The later is the better 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 solution tree computation and to the cost of later trials.

4.2 Attack cost lower bound

A conservative lower bound cost for a successful attack can be computed by assuming that for every added leaf the partial tree is tried without over-cost for the queue to reach the root nor for computing the seed more than once. 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: 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 to find an accepted proof of work.

Proof If we neglect the even distribution of proof leaves, the probability of success at iteration i of constructing a tree (an i-th 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:

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

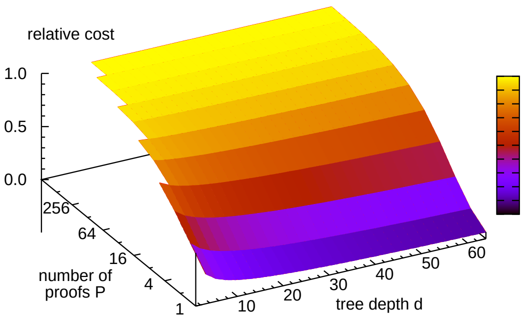

Figure 10: Relative cost lower bound – Equation (2)

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

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

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 $c(\ell) \geq 2\ell + P$ (the ℓ-leaf tree is built and the seed is computed once) 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, as $P \geq 2$:

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

Figure 10 plots this estimation. The back-left corner is empty where the number of proofs is greater than the number of leaves. With $P = 8 \cdot \ln_2(N)$ and if $N \geq 2^7$, Equation (2) is simplified:

$$\mathcal{C}(N) \geq \left(\frac{1}{2}\right)^{\frac{1}{8}} \frac{8 \cdot \ln_2(N)}{8 \cdot \ln_2(N) + 1}(2N) \geq 0.9\,(2N)$$

Namely the average cost for the requester $\mathcal{C}(N)$ is larger than 90% of the $2N$ full tree cost. QED.

4.3 Iterative attack

Let us investigate a simple attack 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: there are $N \cdot f$ leaves in the binary tree, and

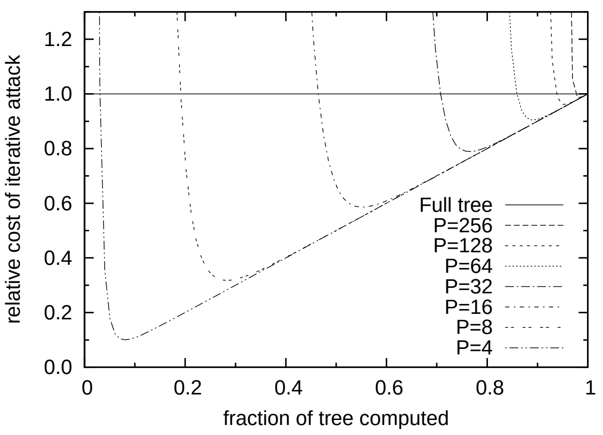

Figure 11: Iterative cost for fraction f with $N = 2^{22}$

about the same number of internal nodes. The second term is the average iteration cost for a solution, by trying faked hash values from depth $\ln_2(P) + 1$ thanks to the even-distribution, and another P to derive the seed from the root hash; the resulting cost is multiplied by the average number of iterations which is the inverse of the probability of success at each trial.

$$\mathcal{C}_{\text{iter}}(f, N, P) \approx 2Nf + (P + \ln_2(P) + 1)\frac{1}{f^P} \qquad (3)$$

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 11 for different number of proofs P: few proofs lead to very beneficial fractions: many proofs make the minimum of the functions close to the full tree computation.

$$\mathcal{F}(N, P) = \sqrt[P+1]{\frac{P\,(P + \ln_2(P) + 1)}{2N}} \qquad (4)$$

Equation (4), the zero of the derivative of (3), 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 Section 4.2. Whether a significantly better strategy can be devised is unclear. A conservative cost lower bound computed with a numerical simulation and for the same parameters gives a 0.961 multiplier. In order to reduce the effectiveness of this attack further, the hash-based generator may cost up to $P \cdot \ln_2(N)$ to derive seed 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, costing at least 90% of the full tree.

4.4 Skewed feedback attack

Let us study the impact of a non-independent proof selection by the pseudo-random number generator. This section simply illustrates the importance of the randomness of the generator. We assume an extreme case where the first bits of the root hash are used as a unique leaf index in the 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 ensure 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 help deter denial-of-service attacks on costly services such as email delivery by requiring moderately hard 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 definitions of optimality criteria: communication-optimal if the minimum amount of data is sent; computation-optimal if the verification is linear in the data sent;

2. a computation-optimal (but not communication-optimal) proof-of-work solution-verification scheme based on Merkle trees with a $e^{\sqrt{w}}$ effort, for which the work on the requester side is bounded and the variance is null: the requester computes $2N$ hashes and communicates $P \ln_2(N)$ data which are verified with $P \ln_2(N)$ computations, with $P = 8 \ln_2(N)$ a good choice;

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 P is sound;

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

These contributions are both theoretical 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 in itself as the verification work is propotional to the data sent by the requester. Also, 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.

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, Ronan Keryell, François Irigoin, John B. Chapman and Zachary B. Ciccolo helped to improve the writing. Comments by anonymous reviewers were also very helpful. Finally, discussions with Gunnar Kreitz and Thomas Sirvent in Casablanca helped to illustrate and clarify this presentation further.

References

[1] Spotify Streaming Protocol. Private communication, 2008. `http://spotify.com`.

[2] 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.

[3] 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, Feb. 2003.

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

[5] 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.

[6] 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.

[7] F. Coelho. An (Almost) Constant-Effort Solution-Verification Proof-of-Work Protocol Based on Merkle Trees. In S. Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, number 5023 in LNCS, pages 80–93. Springer, June 2008.

[8] 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.

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

[10] M. K. Franklin and D. Malkhi. Auditable Metering with Lightweight Security. In *Financial Cryptography 97*, 1997. Updated version May 4, 1998.

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

[12] 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.

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

[14] M. Jakobsson and A. Juels. Proofs of Work and Bread Pudding Protocols. In *Communications and Multimedia Security*, Sept. 1999.

[15] A. Juels and J. Brainard. Client Puzzles: A Cryptographic Defense Against Connection Depletion Attacks. In *Network and Distributed System Security (NDSS)*, Feb. 1999.

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

[17] B. Laurie and R. Clayton. "Proof-of-Work" Proves Not to Work. In *WEAS 04*, May 2004.

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

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

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

[21] R. Rivest and A. Shamir. PayWord and MicroMint – Two Simple Micropayment Schemes. *CryptoBytes*, 2(1), 1996.

[22] 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.

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

[24] 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*, Oct. 2004.

Typeset with LaTeX, revision 851.