PROOFS OF WORK AND BREAD PUDDING PROTOCOLS (EXTENDED ABSTRACT)

Markus Jakobsson
Information Sciences Research Center, Bell Labs, Murray Hill, New Jersey 07974
www.bell-labs.com/user/markusj

Ari Juels
RSA Laboratories, 20 Crosby Drive, Bedford, MA 01730
ari@rsa.com

Abstract

We formalize the notion of a *proof of work* (POW). In many cryptographic protocols, a prover seeks to convince a verifier that she possesses knowledge of a secret or that a certain mathematical relation holds true. By contrast, in a POW, a prover demonstrates to a verifier that she has performed a certain amount of computational work in a specified interval of time. POWs have served as the basis of a number of security protocols in the literature, but have hitherto lacked careful characterization. In this paper, we offer definitions treating the notion of a POW and related concepts.

We also introduce the dependent idea of a *bread pudding protocol*. Bread pudding is a dish that originated with the purpose of re-using bread that has gone stale [18]. In the same spirit, we define a bread pudding protocol to be a POW such that the computational effort invested in the proof may be reused by the verifier to achieve a separate, useful, and verifiably correct computation. As an example of a bread pudding protocol, we show how the MicroMint scheme of Rivest and Shamir can be broken up into a collection of POWs. These POWs can not only serve in their own right as mechanisms for security protocols, but can also be harvested in order to outsource the MicroMint minting operation to a large group of untrusted computational devices.

1. INTRODUCTION

Proof protocols serve as the cornerstone of most algorithms in data security. In a typical cryptographic scenario, one party, the prover, aims to convince another party, the verifier, that it possesses a secret of a certain form, or that a certain mathematical statement holds true. For example, in the Schnorr identification protocol, the prover seeks to demonstrate possession of a secret key corresponding to a specific authenticated public key.

In this paper, we deviate from the standard cryptographic aim of proving knowledge of a secret, or the truth of a mathematical statement. Instead, our goal is to characterize the notion of a *proof of work*, abbreviated POW. This is a protocol in which a prover demonstrates to a verifier that she has expended a certain level of computational effort in a specified interval of time. Although not defined as such or treated formally, POWs have been proposed as a mechanism for a number of security goals, including server access metering, construction of digital time capsules, uncheatable benchmarks, and protection against spamming and other denial-of-service attacks [5, 6, 7, 8, 9, 11, 17].

The contribution of this paper is twofold. First, we offer definitions of the notion of a proof of work (POW) and of related concepts. These definitions are informal; formal definitions will appear in the full version of this paper. As mentioned above, POWs have a demonstrated utility in a number of data security applications. A drawback to their use, however, is the fact that they impose a significant computational load in excess of that associated with many conventional cryptographic protocols. This observation motivates the second contribution of our paper: the idea of *bread pudding protocols*. A bread pudding protocol is based on the same principle as the dish from which it takes its name, namely that of reuse in order to minimize waste. Whereas the traditional bread pudding recipe [18] recycles stale bread, a bread pudding protocol recycles computation. We define a bread pudding protocol to be a POW such that the computational effort invested in the proof may be harvested to achieve a separate, useful, and verifiably correct computation. This idea was first sketched in relation to the anti-spamming technique of Dwork and Naor, who write, "[One] possible scenario would be that in order to send a user a letter, some computation that is *useful* to the recipient must be done. We currently have no candidates for such useful computation" [6].

In this paper, we propose just such a candidate. As an example of a bread pudding protocol, we consider the MicroMint scheme of Rivest and Shamir [16]. We show how the task of minting in this scheme can

be partitioned into a collection of small POWs. These POWs can not only serve in their own right as mechanisms for security protocols, but can also be used to shift the burden of the MicroMint minting operation onto a large group of untrusted computational devices.

The remainder of the paper is organized as follows. In Section 2, we review some of the literature related to POWs. We give definitions of POWs, bread pudding protocols, and related ideas in Section 3. We present our bread pudding protocol for the MicroMint minting operation in Section 4, and conclude in Section 5 with some ideas for future research.

2. PREVIOUS WORK

A number of security protocols in the literature have relied on the use of POWs. Researchers have not previously formalized the notion of a POW, however, and have adopted a wide ranging terminology to describe their constructions.

POWs were perhaps first advocated as a way of attaching a computational cost to resource allocation requests by Dwork and Naor [6]. They propose the use of POWs based on extraction of square roots over prime moduli; on the Fiat-Shamir signature scheme; and on the (broken but still useful) Ong-Schnorr-Shamir signature scheme. POWs in the Dwork and Naor scheme are based on a hash of the time, destination, and the message, and are non-interactive. The sender of a piece of e-mail is required to enclose a POW. Dwork and Naor also introduce the idea of a POW with a trap door, i.e., a function that is moderately hard to compute without knowledge of the secret key, but easy to compute given this key. The availability of trap doors allows designated authorities to generate "postage" without significant expenditure of resources.

A method of controlling spam that may be regarded as an extension of [6] was introduced by Gabber *et al.* [8] (and further explored by Jakobsson and Müller [10]). These proposals also involve use of POWs. The idea is that a server distributes permission – known as a handshake – for a sender to transmit mail to a recipient. This handshake is granted only upon receipt of a valid POW transcript from the sender.

Juels and Brainard [11] propose a related use of POWs as a deterrent against denial-of-service attacks against connection protocols such as SSL. In their scheme, if a malicious party mounts an attack against a server by making many connection requests, the server begins to require clients to perform POWs in order to initiate requests. Their scheme can be extended to non-attack scenarios in which equitable distribution of resources is desired.

POWs have not only been suggested for limiting access, but also to meter it. Franklin and Malkhi [7] describe a scheme that makes use of POWs for third-party verifiable usage metering. A Web site administrator requires users of her site to provide a POW for every access. To demonstrate to an auditor that her site has received a certain amount of usage, she presents the auditor with an audit log consisting of the set of POW transcripts. The auditor verifies the correctness of the audit log. The underlying assumption in this scheme is that many users will have a combined computational power far exceeding that readily available to the site administrator. (Another solution to this problem – but one which does not rely on POWs – was presented in [14].)

Furthermore, POWs have been proposed as a tool to implement *delays*. Rivest *et al.* [17] discuss the creation of digital time capsules, employing a construction which they call a "time-lock puzzle". Their aim is to encrypt data in such a way that the decryption time can be carefully controlled. By discarding the encryption secrets, the data can thus be protected for a period of time designated by the creator. One important feature of the Rivest *et al.* scheme is that the verifier is "implicit" (as defined in Section 3). In particular, a correct POW transcript serves as a decryption key, rather than a means of convincing a verifier. Another distinctive feature is that only feasible way for the prover to complete the POW is sequential and deterministic. This is in contrast to most other POW constructions in the literature, where the prover may use parallel computation. An idea similar to that of time-lock puzzles, namely that of using non-interactive POWs to protect escrowed keys, has been explored in [2, 3, 12].

Another use of POWs to introduce delays was suggested by Goldschlag and Stubblebine [9]. Their aim is to enable verification of the fact that a lottery (or similar application) has been properly administered, while at the same time preventing a premature disclosure of the associated secrets.

Monrose *et al.* [13] propose a method of verifying that a computation is correct by breaking it into multiple pieces and then performing "spot checks" on some random subset of these pieces. Also in this vein is the well established idea of *program checking* (see [4] for a survey). A program checker is a fast program that verifies the correctness of a larger, slower computation. Program checkers have been proposed for a number of program types. Program checkers and other schemes such as that of Monrose *et al.* can both be used directly to create POWs.

Finally, Cai *et al.* [5] propose use of POWs as *uncheatable benchmarks*. These are computational tasks that can effectively be performed in only one way, and enforce a minimum computational load on the

executing entity. Such benchmarks can be crafted in such a way as to prevent vendors from making false claims about the performance of their machines by exploiting computational shortcuts or fine-tuning their language compilers, as is possible with conventional benchmarks. In [5], Cai *et al.* define a benchmark roughly as follows. The benchmark is a task whose performance for the prover represents a problem in some complexity class **P**. Any algorithm in an easier complexity class can solve the task with probability at most 1/4. The verifier possesses a secret s that enables verification of the benchmark as a problem in some complexity class **V**. The complexity class **V** is easier than the complexity class **P**. Thus the prover performs a substantial amount of computation that the verifier can check with relative ease.

This definition of an uncheatable benchmark captures some of the major elements important in the definition of a POW. In particular, it excludes the possibility of a "shortcut" for solving the problem at hand. A correct answer is a demonstration of some minimum expenditure of computational effort. Moreover, this definition imposes the requirement that the verifier be able to check the work of the prover quickly. On the other hand, the definition falls short in several respects of our requirements for a POW. We enumerate some of these here, as they help motivate our own definitions.

- **Precomputation** The attack model elaborated in Cai *et al.* fails to take into account the effect of pre-computation on the part of the prover. Recall that a POW aims to show that a certain amount of computation was performed in a particular interval of time. A prover with sufficient memory resources and time prior to a POW execution, however, can precompute information that makes the POW substantially easier.[1] The effect of precomputation on protocol security is well illustrated by a potential vulnerability in the anti-spamming technique of Dwork and Naor [6]. A POW in their protocol is based on the source and destination addresses of a piece of mail, as well as the current time. An attacker can therefore spend an arbitrarily long period of time pre-computing POWs for a large batch of mail prior to mounting a spamming attack. Our definition of a POW takes pre-computation into account by characterizing the time interval over which the protocol takes place, as well as the memory resources of the prover.

- **Bounds on cheating.** The definition of [5] gives a complexity theoretic characterization of the computation required to perform a POW. This is often not useful for practical security analyses,

such as that in, e.g., [11]. Our definition of a proof of work is more amenable to such analyses, as it establishes more precise bounds.

- **Interleaving.** Cai *et al.* do not characterize the hardness of independent, interleaved POWs. For example, their definition does not exclude the possibility that a prover can successfully execute, e.g., two interleaved POWs as quickly as one. Our definitions account for this possibility.

- **Interactive protocols.** The Cai *et al.* definition can be extended to cover interactive protocols, e.g., protocols involving more than two rounds. Cai *et al.* offer a definition covering a three round protocol. We offer a more general definition allowing an arbitrary number of rounds of interaction.

As mentioned above, we introduce another (and orthogonal) use of POWs known as bread pudding protocols. The aim of this type of protocol is to outsource robustly some of the work associated with a useful computation. This can be done in combination with any of many uses for POWs described in the literature. Thus, completion of a POW in the context of a bread pudding protocol involves a fusion of two aims: the first is to achieve some security goal, such as restricting resource access; the second is to perform what amounts to a computational micropayment, namely the computation harvested by the verifier. The first aim is the conventional one for a POW. The second aim is related in spirit to an idea recently proposed by Ostrovsky [15]. Ostrovsky suggested as an alternative to micropayment schemes the idea of having a client pay for access to a resource by offering a small amount of her computational power. He did not, however, seek to offer any guarantees of correctness/robustness or information hiding. We shall show in our MicroMint scheme an example of how to achieve both correctness/robustness and information hiding in a bread pudding setting. Although we do not offer formal definitions in this extended abstract, the notions of correctness, robustness, and information hiding tie quite naturally into the ideas underlying bread pudding protocols.

3. DEFINITIONS

In this section, we offer a set of definitions enabling us to characterize POWs and their associated properties. Again, in this extended abstract, we offer informal definitions, reserving more precise definitions for exposition in the full version of the paper.

Like any other type of proof protocol, a POW may be either *interactive* or *non-interactive*. Recall that an *interactive proof* is a multi-round

protocol executed by a prover P and a verifier V. In our consideration of POWs, we assume that both P and V may perform an arbitrary number of private coin flips during the protocol execution. At the end of the protocol, V decides either to accept or reject. If V accepts, then the protocol is successful. Otherwise, it has failed. Recall that a *non-interactive proof* involves only one round of communication from the prover. Let c_V denote the private coin flips of V. In order to ensure the security of the proof, it is necessary to generate c_V in a manner that cannot be effectively controlled by the prover. By analogy with non-interactive proofs for standard cryptographic properties, we may accomplish this by reference to a public source of randomness or by some other appropriate means such as, e.g., generating c_V using the hash of some protocol-specific value. Thus, in a non-interactive proof protocol, the prover simulates a communication from the verifier, and then sends its transcript to the verifier.

An important variant on these ideas is that of an *implicit* POW. An implicit POW is a type of non-interactive proof protocol in which verification is not performed by a verifier, but is determined by the ability of the prover to perform a given task. For example, a correct POW transcript can serve as a decryption key for some escrowed key or document, as in, for example, [12] or [17]. Thus the prover or any other party is capable of verifying a correct implicit POW without the active participation of the verifier. As an example of an implicit POW, we briefly describe in Section 1 the notion of a time-lock puzzle, as proposed by Rivest *et al.*

Let us assume in our definitions, for the sake of simplicity, that no communications latency is incurred in a POW. (Our definitions can be modified to accommodate communications latency as appropriate.) We define the *start time* t_s of the protocol to be the time at which the verifier initiates its first round of communication. The *completion time* t_c is the time at which the last round of the protocol is complete. The aim of a POW is to enable P to demonstrate that she has performed a certain amount of computation within the time interval $[t_s, t_c]$. Let $poly$ denote any polynomial in a given variable. (We use the informal notation $o(1/poly(x))$ to denote a quantity that is asymptotically smaller than the inverse of any polynomial in x.) Finally, let l be a security parameter. Finally, let us assume that the prover is permitted to perform an arbitrarily large amount of computation prior to the protocol execution. Thus, in fact, our definitions assume that the prover may perform computation over the time interval $[-\infty, t_c]$. We characterize the hardness of a POW using the following two definitions, where probabilities are over the coin flips of both parties, and a computational step

is as measured in any suitable model. Definition 1 provides the notion of a lower bound on POW hardness, while definition 2 provides that of an upper bound.

Definition 1 *We say that a proof of work POW is* (w, p)-hard *if the following is true. For any prover P with memory resources bounded by m, if P performs at most w steps of computation in the time interval $[t_s, t_c]$, then the verifier V accepts with probability at most $p + o(\frac{m}{poly(l)})$.*

Definition 2 *We say that a proof of work POW is* (w, p, m)-feasible *if there exists a prover P with memory resources bounded by m such that after w steps of computation in the time interval $[t_s, t_c]$, the prover can cause the verifier V to accept with probability at least p.*

This leads to the following definition. Note that we set $m = w$ (somewhat arbitrarily) as an upper bound on the memory available to the prover as this is the maximum amount of memory the prover can use in w computational steps. Note also that it is possible to relax both this and the next definition to allow for, e.g., $(w, 1 - \epsilon, w)$-feasibility, where ϵ is quantity negligible with respect to the security parameter l. For the sake of simplicity, we do not consider such definitional variants.

Definition 3 *We say that a proof of work POW is* sound*, if, for some w, POW is $(w, 1, w)$-feasible.*

A POW may be regarded as efficient if the verifier performs substantially less computation than the prover. In keeping with the definition of Cai *et al.* [5], we say that such a proof has a large *advantage*, defined as follows.

Definition 4 *Let POW be a sound proof of work, and w be the minimum value such that POW is $(w, 1, w)$-feasible. Let z be the maximum amount of computation performed by the verifier on a correct transcript*[2] *for POW. The* advantage *of POW is equal to w/z.*

Recall that one of the aims of our definitions is to consider whether it is possible for a prover to "cheat" somehow on batches of POWs. In particular, we consider whether it is possible for the prover to perform multiple, possibly interleaved proofs of work successfully with less computation than that required for performing the POWs individually. This leads us to define the notion of *independence* on POWs. Our definition ensures that independent POWs are not vulnerable to prover cheating in the form of batch processing.

Definition 5 *Let POW_1 and POW_2 be two proofs of work for which the respective coin flips of the verifier are generated independently. Let POW' be a proof of work constructed by combining (possibly interleaving) POW_1 and POW_2. In other words, the verifier accepts for POW' if it accepts for POW_1 and for POW_2. We say that POW_1 and POW_2 are* independent *if the following is true. If POW' is (w, p, m)-feasible, then for some w_1, w_2, p_1, and p_2 such that $w = w_1 + w_2$ and $p = p_1 p_2 + o(m/poly(l))$, it is the case that POW_1 is (w_1, p_1, m)-feasible and POW_2 is (w_2, p_2, m)-feasible.*

The final definition we present here relates to the notion of a *bread pudding protocol*.

Definition 6 *Suppose that POW_1 is a (w, p)-hard proof of work. Let P_1 denote the prover involved in this proof of work, and V_1 the corresponding verifier. Suppose that P_1 is also a verifier (denoted V_2) in a proof of work POW_2, for which the prover is denoted P_2. We say that POW_2 is a* bread pudding protocol *for POW_1 if the following is true. If $P_1 (= V_2)$ accepts the transcript for POW_2, then P_1 can perform w computational steps over the duration of POW_1 and convince V_1 to accept its transcript with probability at least $p + 1/poly(l)$.*

In this definition, we see that the computation that P_2 performs in POW_2 is recycled for use in POW_1. In a sense, we may regard POW_2 as an oracle for POW_1. A bread pudding protocol POW_2 is one in which this oracle reduces the computational burden of prover P_1 in the POW_1. If POW_1 is an implicit bread pudding protocol, then POW_2 may be viewed as helping to solve a computational problem, rather than aiding in successful completion of an interactive POW. Of course, trivially, if $POW_2 = POW_1$, then POW_2 is a bread pudding protocol for POW_1. In order for POW_2 to be of interest as a bread pudding protocol, it must have additional properties, such as robustness, or information hiding (see, e.g., [1]) or divisibility, i.e., the ability to generate independent copies such that it is possible to derive useful work from multiple, independent provers. Due to lack of space, we do not explore the definitions of these properties in this extended abstract. Our bread pudding protocol for MicroMint, however, has all of them, as we shall see.

3.1 EXAMPLE OF A POW

In order to make our definitions more concrete, we now present an example of a POW. This POW is very similar to that employed in several proposed security protocols, including those in [8, 11]. It is also similar to the basis of our bread pudding protocol for MicroMint in Section 4.

This POW, which we call a *partial inversion proof of work* (PIPOW), requires two rounds.

Example 1 (PIPOW) *Let h represent a one-way function. The verifier V generates a random bitstring x of length l and computes the image $y = h(x)$. Let x' be the first $l - k$ bits of x, where $k \leq l$. V sends the pair (x', y) to P. In order to complete the POW successfully, P must calculate a valid pre-image $\tilde{x}$ of y.*

It is easy to see that PIPOW is $(w, 1/(2^k - w), O(1))$-feasible for any integer $w \in [0, 2^k - 1]$. The hardness associated with this POW may be characterized by the following claim (which states that the observed feasibility is a tight). We defer a proof of this claim, to be given in the random oracle model on h, for the full version of this paper.

Claim 1 *PIPOW is (w, p)-hard for any integer $w \in [0, 2^k - 1]$ and $p = 1/(2^k - w)$.*

4. BREAD PUDDING FOR MICROMINT

As an example of a bread pudding protocol, we consider the highly computationally intensive operation of minting in the MicroMint scheme. We show how to partition this task into a collection of POWs, enabling minting to be distributed among a collection of low power, untrusted entities. This is done without allowing the successful collisions (corresponding to micro-coins) to be "stolen" by the prover. Thus the bread pudding protocol we elaborate for MicroMint has the properties of information hiding, divisibility, and robustness. The associated POWs are also independent, under certain assumptions about the underlying hash function. Let us begin by describing how MicroMint works.

4.1 MICROMINT

MicroMint is a micropayment system developed by Rivest and Shamir [16]. Its security is based on the hardness of finding hash function collisions. A coin in this scheme consists of a k-way hash function collision, that is, a set $\{x_1, x_2, \ldots, x_k\}$ of pre-images that map to a single image. A number of variants and deterrents, which we do not describe here, create strong deterrents to coin theft and double-spending.

The security of MicroMint against forgery derives from the large base computational costs associated with the minting operation. With appropriate parameterization of the scheme, minting a single coin is difficult, while the marginal cost associated with minting many coins is relatively small. (The use of k-way collisions, rather than 2-way collisions, increases the computational threshold required for producing the

first coin.) Thus, minting requires a substantial base investment in hardware. For forgery to be successful, it must take place on too large a scale to make the effort worthwhile. By limiting the period of validity of a given coin issue and computing the issue over an extended period of time, the minter can even make the job of a forger harder than his own.

Suppose that the hash function h used for minting maps n-bit pre-images to n-bit images. The process of finding collisions may be thought of as that of throwing balls uniformly at random into a set of 2^n bins. Throwing a ball corresponds in this model to choosing a pre-image x and placing it in the bin with index $h(x)$. When k balls land in a single bin, they together constitute a coin.

If n is to be large enough to ensure an adequate level of security, the storage overhead associated with maintaining 2^n bins will be prohibitively large. Rivest and Shamir thus describe the following variation on the basic scheme. Let $n = t + u$. A ball (pre-image) x is considered valid only if the t least significant bits of $h(x)$ match some pre-selected, random value s. (If invalid, the ball may be considered to miss the set of bins.) A valid ball is thrown into one of a set of 2^u bins, according to the value of the u most significant bits. With the right parameterization, the computational effort associated with minting is still high, but the number of bins is smaller. Rivest and Shamir recommend, heuristically, the use of $k2^u$ balls.

Note that to prevent a potential forger from initiating her effort prior to a given coin issue, it is possible in Rivest and Shamir's original scheme to key the hash function h with a secret value r that is only released on the issue date. For additional details and descriptions of a number of variants, the reader is advised to see [16].

4.2 BREAD PUDDING MINTING

We now demonstrate a simple bread pudding protocol for MicroMint, that is, a MicroMint variant in which the computation associated with minting may be embodied in a set of POWs. This bread pudding protocol has robustness, independence, and information hiding properties, as we shall see.

Let h be a suitable hash function and $\|$ denote string concatenation. We define a ball to be a triplet (i, x, y), where $y = h(r \parallel i)$ and r is a secret value as above. A valid ball is one in which the t least significant bits of $h(x \parallel y)$ are equal to s. The bin into which a ball is thrown is determined by the u most significant bits of $h(x \parallel y)$.

The computational cost associated with minting in this MicroMint variant remains the same as in the original scheme. Verifying the va-

lidity of a coin in the variant requires twice the number of hashes as the original. The advantage of the variant scheme, however, is that the problem of finding a single, valid ball may be distributed as a POW. By distributing enough of these POWs, the minter may offload the majority of the computation associated with the minting operation.

In this scheme, the verifier initiates the POW by sending to the client (prover) the pair (i, y), where i is a session identifier or other protocol-specific value, and $y = h(r \parallel i)$. The verifier also sends the parameter pair (s, t). The task of the client is to find a value x such that the first t bits of $h(x \parallel y)$ are equal to s, i.e., such that triple (i, x, y) is a valid ball. The client returns this triple to the verifier (assuming that the verifier, to achieve statelessness, does not store it). This POW requires an average computational effort of 2^{t-1} hashes for the prover. In fact, it may be seen that this is a $(w, 1/(2^t - w), O(1))$-feasible and also a $(w, 1/(2^t - w))$-hard POW, in accordance with the definitions and example in Section 3. The verifier requires two hashes to check the validity of a ball: one hash to verify that $y = h(r \parallel i)$, where, again, r is the secret minting value, and one hash to verify that the first t bits of $h(x \parallel y)$ are equal to s.

Note that the secret value r is not revealed in a POW. Thus, even when minting is performed by way of POWs, this secret value need only be released on the day of coin issue. This information hiding ensures that security comparable to that of the original scheme is achieved. In particular, an adversary sees only the valid balls that he himself computes or which he has access to through colluding parties. Unless he can collect the vast majority of valid balls, though, the minting operation remains infeasible for him. In particular, it is infeasible for him to obtain r and create new balls. Observe also, that the POWs in this scheme, assuming that h has random-oracle like properties, are independent. Due to constraints of space, we omit a formal analysis of the information hiding and other security properties of this scheme.

Rivest and Shamir propose sample parameters in their paper of $k = 4$, $n = 52$, and $t = 21$ for achieving a viable minting setup. Thus, the POW based on finding a valid ball requires an average of 2^{20} hash operations for the prover. This is, as it happens, exactly the hardness of the POW proposed in [8], requiring about 2 seconds on a 266 MHz Pentium II processor under the hash function MD5. If the minter offloads the problem of finding valid balls onto clients, then his own computational effort is equal to at most three hashes per ball: two for verification, one to determine which bin a given ball belongs in. Given the number $k2^u = 2^{33}$ of balls suggested by the heuristic calculations in [16], the minter would thus have to perform roughly 2^{35} hash function computations. This can be computed in less than a day on a standard desktop computer, with

sufficient available memory. Without outsourcing the minting operation, the minter would be forced to perform roughly 2^{53} hash function computations on average.

Altogether, a set of 2^{33} POWs requiring an average of 2 seconds of computation apiece represents a substantial amount of computation to offload onto clients. With one million clients, for instance, it would be necessary for each client to perform almost five hours of computation to complete the solution of all POWs. In many cases – as when clients can perform computation overnight using idle cycles – this is reasonable. Nonetheless, in some cases, as when clients are very low power devices, it may be desirable to make the POWs somewhat easier. We can do this as follows. Let us require that y in a valid ball have v leading $'0'$ bits, and that only the first $t - v$ bits in $h(x \parallel y)$ be equal to a value s. Now a POW requires only 2^{t-v-1} hash computations on average for a client. A POW, of course, is harder for the minter in this case: the minter effectively compensates for the reduced computational burden on clients by performing substantially more computation itself. The memory requirements in this variant of our scheme, however, are unchanged.

5. CONCLUSION: SOME OPEN PROBLEMS

We conclude by offering brief mention of some open problems motivated by this paper. The first of these is the problem of devising other useful bread pudding protocols. Other examples of bread pudding protocols would be desirable not only in themselves, but perhaps as a step toward defining a large class of computational problems amenable to partitioning into POWs. Another open problem relates to proving results about the hardness of POWs. We offer in this paper a proof in the random oracle model of the hardness associated with a common type of POW based on hash function inversion. Of use would be a stronger result more precisely characterizing the required properties of hash functions for this purpose. This line of exploration might yield additional results. For example, since POWs generally involve only a few seconds of computation, it seems likely that weak cryptographic functions would serve in lieu of the conventional strong ones. This might yield more efficient POWs. It might also have the interesting incidental consequence (as in [6]) of furnishing a means of re-using for certain cryptographic algorithms that have been broken in a conventional sense – in keeping with the spirit of reuse that motivates bread pudding.

Acknowledgments

We wish to thank Burt Kaliski and Julien Stern as well as the anonymous reviewers of this paper for providing references and suggestions.

Notes

1. Trivially, even a slow prover can, with an arbitrarily large amount of time and memory, pre-compute all possible proof-of-work transcripts and therefore complete a POW instantaneously. For a benchmark, this is particularly problematic, as it means that the prover can convince the verifier that he has an arbitrarily large amount of computing power.

2. We consider the number of steps of computation performed by the verifier on a correct transcript, as the verifier can always terminate its computation after this many steps.

References

[1] M. Abadi, J. Feigenbaum, and J. Kilian. On hiding information from an oracle. *Journal of Computer and System Sciences*, 39(1):21–50, Aug 1989.

[2] M. Bellare and S. Goldwasser. Encapsulated key escrow. Technical Report Technical Report 688, MIT Laboratory for Computer Science, April 1996.

[3] M. Bellare and S. Goldwasser. Verifiable partial key escrow. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 78–91, April 1997.

[4] M. Blum and H. Wasserman. Software reliability via run-time result-checking. *Journal of the ACM.* To appear. Preliminary version: 'Program Result-Checking: A Theory of Testing Meets a Test of Theory,' Proc. 35th IEEE FOCS, 1994, pp. 382-392.

[5] J. Cai, R. Lipton, R. Sedgewick, and A. Yao. Towards uncheatable benchmarks. *IEEE Structures*, pages 2–11, 1993.

[6] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Proc. CRYPTO '92*, pages 139–147. Springer-Verlag, 1992. Lecture Notes in Computer Science No. 740.

[7] M.K. Franklin and D. Malkhi. Auditable metering with lightweight security. In R. Hirschfeld, editor, *Proc. Financial Cryptography '97 (FC '97)*, pages 151–160. Springer-Verlag, 1997. Lecture Notes in Computer Science No. 1318.

[8] E. Gabber, M. Jakobsson, Y. Matias, and A. Mayer. Curbing junk e-mail via secure classification. In R. Hirschfeld, editor, *Financial Cryptography '98.* Springer-Verlag, 1998.

[9] D. Goldschlag and S. Stubblebine. Publicly verifiable lotteries: Applications of delaying functions. In R. Hirschfeld, editor, *Financial Cryptography '98*. Springer-Verlag, 1998.

[10] M. Jakobsson and J. Müller. How to defend against a militant spammer, 1999. manuscript.

[11] A. Juels and J. Brainard. Client puzzles: A cryptographic defense against connection depletion attacks. In *Proceedings of the 1999 ISOC Network and Distributed System Security Symposium*, pages 151–165, 1999.

[12] S. Micali. Guaranteed partial key escrow. Technical Report Technical Memo 537, MIT Laboratory for Computer Science, September 1995.

[13] F. Monrose, P. Wyckoff, and A. Rubin. Distributed execution with remote audit. In *Proceedings of the 1999 ISOC Network and Distributed System Security Symposium*, pages 103–113, 1999.

[14] M. Naor and B. Pinkas. Secure and efficient metering. In K. Nyberg, editor, *Advances in Cryptology – Eurocrypt '98*, pages 576–590. Springer-Verlag, 1998.

[15] R. Ostrovsky. A proposal for internet *computational* commerce: How to tap the power of the WEB, 1998. Presentation at CRYPTO '98 Rump Session.

[16] R.L. Rivest and A. Shamir. PayWord and MicroMint–two simple micropayment schemes. RSA Laboratories. *CryptoBytes*, 2(1):7–11, Spring 1996.

[17] R.L. Rivest, A. Shamir, and D. Wagner. Time-lock puzzles and timed-release crypto. To appear, 10 March 1996.

[18] Irma S. Rombauer and Marion Rombauer. Bread-pudding with meringue (six servings). In *Joy of Cooking*, page 751. Penguin Group, 1997.