Hash Cash - Amortizable Publicly Auditable Cost Functions

Adam Back
e-mail: adam@cypherspace.org

9th July 2001

Abstract

We present a distributed efficiently amortizable CPU cost function with no trap–door. The absence of a trap–door allows us to avoid needing to trust any party.

Applications for such cost functions are in distributed document popularity estimation, and metering of web advertising. None of the servers involved have any advantage over users in computing the cost function.

The amortized token has a small fixed sized representation independent of the amortized value. The valuation function is efficient.

Limits can be placed on the resources which can be expended in computing the cost function to prevent the user inflating the value of his contribution by using more CPU time than the expected token value.

The amortized part of the cost result can be blinded so that clients can not obtain service but avoid contributing to the amortization by expending more resources than the expected token value.

Interactive and non-interactive variants of the cost function can be constructed which can be used in situations where the server can issue a challenge (connection oriented interactive protocol), and where it can't (where the communication is store–and–forward, or packet oriented) respectively.

Key Words: hashcash, cost functions, client puzzles, distributed metering, distributed popularity estimation

1 Introduction

CPU cost functions have been proposed by Dwork and Naor in [2], and by Back in [1] to limit systematic abuse of unmetered internet resources such as Internet email, and anonymous remailers. Franklin and Malkhi in [3] propose a (non–amortizable) distributed web usage metering scheme based on incremental cost functions. Juels and Brainard in [4] explore the application of cost functions to frustrating the connection depletion attack – a class of denial of service attack on connection oriented protocols such as TCP/IP and SSL/TLS.

2 Cost Functions

A *cost function* should be computationally cheap to verify, but parameterisably expensive to compute. We use the following notation to define a cost function.

In the context of cost functions we use *client* to refer to the user who must compute a *token* (usually denoted t) using a cost function MINT() which is used to create tokens to participate in a protocol with a *server*. We use the term *mint* for the cost function because of the analogy between creating cost tokens and minting physical money.

The server will check the value of the token using an evaluation function VALUE(), and only proceed with the protocol if the token has the expected value.

The functions are parameterised by the amount of work w that the user will have to expend on average to mint a token.

With *interactive cost functions*, the server issues a challenge c to the client – the server uses the $\mathsf{CHAL}()$ function to compute the challenge. (The challenge function is also parameterised by the work factor.)

$$\begin{cases} c = \mathsf{CHAL}(w) & \text{server challenge function} \\ t = \mathsf{MINT}(c) & \text{mint token based on challenge} \\ v = \mathsf{VALUE}(t) & \text{token evaluation function} \end{cases}$$

With *non-interactive cost functions* the client choses it's own challenge or random start value in the MINT() function, and there is no CHAL() function.

$$\begin{cases} t = \mathsf{MINT}(w) & \text{mint token} \\ v = \mathsf{VALUE}(t) & \text{token evaluation function} \end{cases}$$

Clearly a *non-interactive cost function* can be used in an interactive setting, whereas the converse is not possible.

2.1 Known solution

- A ***known solution*** cost function is a cost function where the server knows or can cheaply compute the result of the challenge it issues.

Where we can rely on an interactive protocol, the challenger can issue the client with a challenge with a single known solution. A simple way to create known solution cost functions is to base them on a trap-door problem such as the factoring problem.

$$\begin{cases} \mathsf{PUBLIC:} & n = pq \\ \mathsf{PRIVATE:} & \text{primes } p \text{ and } q,\ \phi(n) = (p-1)(q-1) \\ \\ c = \mathsf{CHAL}(w) & \textbf{choose } r \in_R [0, n) \\ & \textbf{return } (r, w) \\ t = \mathsf{MINT}(c) & \textbf{compute } x = r^{r^w} \pmod n \\ & \textbf{return } (x, w) \\ v = \mathsf{VALUE}(t) & \textbf{if } r^{r^w} = t \pmod n \textbf{ return } w \\ & \textbf{else return } 0 \end{cases}$$

The client does not know $\phi(n)$, and so the most efficient method for the client to calculate MINT() is repeated exponentiation, which requires w exponentiations. The challenger knows $\phi(n)$ which allows a more efficient computation of $r^{r^w} \bmod n$, namely reducing $r^w \bmod \phi(n)$, so the challenger can execute VALUE() with 2 modular exponentiations. The challenger as a side-effect has a trapdoor in computing the cost function as he can compute MINT() efficiently using the same algorithm. (This cost function is related to a construct Rivest et al used for *time-lock puzzles* in [5].)

2.2 Publicly Auditable, Probablistic Cost

- A ***publicly auditable*** cost function can be *efficiently* verified by any third party without access to any trapdoor or secret information. (When we say *publicly auditable* we mean implicitly that the cost function is *efficiently* publicly auditable, all cost functions are publicly auditable by definition, as the auditor can just repeat the work done by the client.)

- A ***fixed cost*** cost function takes a fixed amount of resources to compute. The fastest algorithm to mint a fixed cost token is a deterministic algorithm. The factoring based cost function given above is *fixed cost*.

- A ***probablistic cost*** cost function is one where the cost to the client of minting a token has a predictable expected time, but a random actual time as the client can most efficiently compute the cost function by starting at a random start value. Sometimes the client will get lucky and start very close to the solution.

  There are two types of probablistic cost *bounded probablistic cost* and *unbounded probablistic cost*.

  – An ***unbounded probablistic cost*** cost function, can in theory take forever to compute, though the probablity of taking significantly longer than expected shrinks logarithmically. (An example would be the cost function of being required to throw a head with a fair coin; in theory the user could be unlucky and end up throwing thousands of tails, but in practice the probability of not throwing a head for k throws tends towards 0 rapidly as $\lim_{k\to\infty} \frac{1}{2}^k = 0$.)

– With a ***bounded probablistic cost*** cost function there is a limit to how unlucky the client can be in it's search for the solution; for example where the client is expected to search some key space for a known solution; the size of the key space imposes an upper bound on the cost of finding the solution.

Juels and Brainard in [4] give a cost function they call a *client puzzle*.

Client puzzles are interactive, known solution, trapdoor, cost functions with bounded probablistic cost. Client puzzles are *not* publicly auditable, because private information is required to verify them. (Note the secret key material used in the client puzzle Juels and Brainard propose is introduced only as an optimization to avoid needing to keep state. If this optimization were removed the resulting cost function would be *publicly auditable*.)

Client puzzles retain the trapdoor property, in that the server can cheaply mint tokens, although the ability to cheaply mint tokens does not derive from private key material. The trapdoor property derives from the fact that client puzzles are interactive, known solution cost functions: the server knows the solution to the puzzle at the time it creates the puzzle.

First we introduce some notation: consider bitstring $s = \{0,1\}^\star$, we define $[s]_i$ to means the bit at offset i, where $[s]_1$ is the left-most bit, and $[s]_{|s|}$ is the right-most bit. $[s]_{i\ldots j}$ means the bit-wise substring between and including bits i and j, $[s]_{i\ldots j} = [s]_i \parallel \ldots \parallel [s]_j$. So $s = [s]_{1\ldots|s|}$.

The client puzzle cost function is:

$$\left\{\begin{array}{ll}
\mathsf{PUBLIC}: & \text{hash function } h() \text{ with output size } k \text{ bits} \\
\mathsf{PRIVATE}: & \text{server seed key } s \\
 & \\
c = \mathsf{CHAL}(w) & \textbf{choose } r \in_R \{0,1\}^k \\
 & \textbf{compute } x = h(s\|r\|w) \\
 & \textbf{compute } y = h(x) \\
 & \textbf{set } p = [x]_{w+1\ldots k} \\
 & \textbf{return } (p,y,r,w) \\
t = \mathsf{MINT}(c) & \textbf{find } z \in_R \{0,1\}^w \textbf{ st } y = h(z\|p) \\
 & \textbf{set } x' = z\|p \\
 & \textbf{return } (x',y,r,w) \\
v = \mathsf{VALUE}(t) & \textbf{if } y = h(x') \textbf{ and } x' = h(s\|r\|w) \textbf{ return } w \\
 & \textbf{else return } 0
\end{array}\right.$$

Juels and Brainard also give a method for reducing the *variance* of the bounded probablistic cost of their client puzzles. The technique is to compose each puzzle of a number of sub puzzles such that the expected cost remains the same. This property can be useful in interactive settings where you don't want the user to experience large variance in latency.

2.3 Trapdoor-free

A disadvantage of known solution cost functions is that the challenger can cheaply create tokens of arbitrary value. This precludes public audit where the server may have a conflict of interests, for example in web hit metering, where the server may have an interest to inflate the number of hits on it's page where it is being paid per hit by an advertiser.

- A ***trapdoor-free*** cost function is one where the server has no advantage in minting tokens.

An example of a trapdoor-free cost function is the Hashcash [1] cost function.

Hashcash is a non-interactive, publicly auditable, trapdoor-free cost function with unbounded probablistic cost.

We define a pair of binary infix comparison operators $\overset{\text{left}}{=}_b$ where b is the length of the common left-substring from the two bit-strings, and $\overset{\text{right}}{=}_b$ is the length of the common right-substring.

$$\begin{array}{ll}
x \overset{\text{left}}{=}_0 y & [x]_1 \neq [y]_1 \\
x \overset{\text{left}}{=}_b y & \forall_{i=1\ldots b}\ [x]_i = [y]_i
\end{array}$$

$$\begin{array}{ll}
x \overset{\text{right}}{=}_0 y & [x]_{|x|} \neq [y]_{|y|} \\
x \overset{\text{right}}{=}_b y & \forall_{i=1\ldots b}\ [x]_{|x|+1-i} = [y]_{|y|+1-i}
\end{array}$$

Hashcash is computed relative to a service-name s, to prevent tokens minted for one server being used on another (servers only accept tokens minted using their own service-name). The service-name can be any bit-string which uniquely identifies the service (eg. host name, or email address).

The hashcash function is defined as:

$$\begin{cases} \mathsf{PUBLIC}: & \text{hash function } h() \text{ with output size } k \text{ bits} \\ \\ t = \mathsf{MINT}(s, w) & \textbf{find } x \in_R \{0,1\}^\star \textbf{ st } h(s) \stackrel{\text{left}}{=}_w h(s\|x) \\ & \textbf{return } (x, s) \\ v = \mathsf{VALUE}(t) & h(s) \stackrel{\text{left}}{=}_v h(s\|x) \\ & \textbf{return } v \end{cases}$$

The hashcash cost function is based on finding *partial hash collisions*. The fastest algorithm for computing partial collisions is brute force. There is no challenge as the client can safely choose his own random challenge, and so the hashcash cost function is a *trapdoor-free* and *non-interactive* cost function.

In addition the Hashcash cost function is *publicly auditable*, because anyone can efficiently verify any published tokens.

(In practice $|x|$ should be chosen to be large enough to make the probability that clients collectively reuse a previously used start value negligible; $|x| = 128$ bits should be enough even for a busy server.)

The server needs to keep a double spending database of spent tokens, to detect and reject attempts to spend the same token again. To prevent the database growing indefinately, the service string can include the time at which it was minted. This allows the server to discard entries from the spent database after they have expired. Some reasonable expiry period should be chosen to take account of clock inaccuracy, computation time, and transmission delays.

Hashcash was originally proposed as a counter-measure against spam in email, and against systematic abuse of anonymous remailers. It is necessary to use *non-interactive cost functions* for these scenarios as there is no channel for the server to send a challenge over. However one advantage of *interactive cost functions* is that it is possible to prevent pre-computation attacks. For example, there is a cost associated with sending each email, which tends to discourage email abuse; however a determined adversary may spend a year pre-computing tokens to all be valid on the same day, and on that day be able to overload the system.

It would be possible to mitigate the pre-computation attack by using a slowly changing *beacon* (unpredictable broadcast authenticated value changing over time) such as say this weeks winning lottery numbers. In this event the public value is included in the start string, limiting pre-computation attacks to being conducted within the time period between beacon value changes.

2.4 Non-interactive and bounded probabilistic cost

The following cost function achieves bounded probabilistic cost in the non-interactive setting by using a *kosherized* (verifiably fairly chosen) choice of random permutation with chosen permutation size. Blowfish is given as an example of a block cipher which can be readily given a reduced keyspace to provide a reduced sized fair random permutation. It's s-boxes and key are chosen using an efficient kosherized algorithm. It is necessary to kosherize the selection of the s-boxes, otherwise the client may find a way to maliciously select an s-box defining a family of permutations which he can compute the desired permutation invariant more efficiently than by brute force. It is necessary to choose a different random s-box fairly for each token, because the reduced key space would make using the same s-box, or related s-boxes open to pre-computation attacks.

We introduce a naming scheme for blowfish variants: $\mathsf{BF\text{-}KDF}(s)\text{-}b$, where KDF is a key derivation function used with seed s to select kosherized (verifiably fair) s-box values. b is the blocksize, and must be a multiple of 8-bits. So standard blowfish would be denoted $\mathsf{BF\text{-}DIGITS}(\pi\text{-}3)\text{-}64$, where $\mathsf{DIGITS}()$ is a KDF which returns succesive bits from the binary fraction representation of the floating point seed value.

Then we define a family of reduced round kosherized s-box blowfish variants: $\mathsf{BF\text{-}SHA1KDF}(s)\text{-}b$, where $\mathsf{SHA1KDF}(s)$ outputs $\mathsf{SHA1}(i\|s)$ on it's i-th invocation. The 18 p-array and $(\frac{b}{8})^2$ s-box $\frac{b}{2}$-bit values are selected from the output of the KDF.

What invariant can we use for this?

2.5 Non-interactive and fixed cost

The following cost function achieves fixed cost in the non-interactive setting by using a kosherized choice of random permutation and key with reduced keyspace.

For this we need a single solution problem, which requires search of the entire key space, no matter where the client chooses to start, and which is efficiently verifiable.

The permutation invariant is to find (hmmm... think about this).

DES invariant they used deepcrack to find? Will this invariant work for reduced key space DES? Can DES's keyspace be reduced efficiently? (Is there an efficient secure fair s-box selection algorithm, or at least fast enough for the reduced keyspaces needed?)

Other generic permutation invariant?

2.6 Amortizable

- A ***probabilisticly amortizable*** token is one where there exists an efficient algorithm to combine tokens into a single token with workably compact representation with value which probabilisticly approximates the combined value. Repeatedly combining the same tokens again or combining the token with itself should not increase it's value.
- A ***publicly amortizable*** token is one where the amortization function does not require any private information.
- A ***publicly auditable amortizable*** token is one where the amortized token's value is efficiently verifiable by anyone without any private information.

Non-interactive Amortizable Hashcash is a non-interactive, publicly auditable, trapdoor-free, unbounded probablistic cost, probabilisticly amortizable cost function.

The amortizable hashcash function is defined:

$$\begin{cases} \mathsf{PUBLIC}: & \text{hash function } h() \text{ with output size } k \text{ bits} \\ t = \mathsf{MINT}(s, w) & \textbf{find } x \in_R \{0,1\}^\star \textbf{ st } h(s) \overset{\text{left}}{=}_w h(s \| x) \\ & \textbf{return } (s, x) \\ v = \mathsf{VALUE}(t) & h(s) \overset{\text{left}}{=}_v h(s \| x) \\ & \textbf{return } (v) \\ t = t_1 + t_2 & \textbf{if } \mathsf{VALUE}(t_1) > \mathsf{VALUE}(t_2) \textbf{ return } (t_1) \\ & \textbf{else return } (t_2) \end{cases}$$

For applications such as web metering, the Non-interactive Amortizable Hashcash protocol has a weakness in that the protocol allows forms of cheating called *under-contributing* and *over-contributing* by clients. With *under-contributing* the client spends some extra computation to ensure that he minimally complies with the token value required, to avoid contributing to the amortized value.

The converse client action of *over-contributing* may sometimes be considered an attack, depending on the use. For web metering, where the amortizated token is intended to measure hits served by a web hosting service, over-contributing could be a problem, as users have the ability to inflate the measurement of web-hits. Over-contributing as such can not be prevented, as the user can simply request the document multiple times. But there remains some value to defending against over-contribution, as the user has to download the whole document which is an additional cost for the user (and the hosting service), and the server can to some extent detect and refuse multiple downloads by the same user.

For distributed document popularity where the estimate of hits served is not relevant, over-contributing may be encouraged as a more bandwidth efficient way for users to rate documents.

- A ***fair amortizable*** cost function is one where the client can not *under-contribute*. *Over-contributing* may optionally be allowed.

To simultaneously achieve both document popularity metric collection and amortizable document hit statistics we would use a pair of tokens: a non-interactive amortizable token for document popularity, and a fair amortizable token for hit statistics. Both tokens would be independently amortized.

Interactive Fair Amortizable Hashcash is an interactive, publicly auditable, trapdoor-free, unbounded probabilistic cost, fair probabilisticly amortizable cost function.

The Interactive Fair Amortizable Hashcash function is defined below.

$$
\begin{cases}
\mathsf{PUBLIC}: & \text{hash function } h() \text{ with output size } k \text{ bits} \\
c = \mathsf{CHAL}(s, w) & \textbf{choose } b' \in_R \{0,1\}^\star \\
 & \textbf{compute } b = h(s \| b') \\
 & \textbf{return } (s, b, w) \\
t' = \mathsf{MINT}(c) & \textbf{find } x \in_R \{0,1\}^\star \textbf{ st } b \overset{\text{left}}{=}_w h(s \| b \| x) \\
 & \textbf{return } (s, b, w, x) \\
t = \mathsf{UNBLIND}(t') & \textbf{return } (s, b', w, x) \\
v = \mathsf{VALUE}(t) & \textbf{compute } b = h(s \| b') \\
 & \textbf{if } b \overset{\text{left}}{\neq}_w h(s \| b \| x) \textbf{ return } 0 \\
 & h(s \| b \| x) \overset{\text{right}}{=}_u b' \\
 & \textbf{return } (w + u) \\
t = t_1 + t_2 & \textbf{if } \mathsf{VALUE}(t_1) > \mathsf{VALUE}(t_2) \textbf{ return } (t_1) \\
 & \textbf{else return } (t_2)
\end{cases}
$$

Some notes on the design criteria for Fair Amortizable Hashcash:

- The cost of the verification function is minimised. The verification function costs 2 $h()$ operations.
- The server should not be able to trick users into computing tokens on other service-names. s is included in $h(s\|b\|x)$ for this reason even though the service-name s will later become apparent when b is unblinded as $b = h(s\|b')$.
- There should be no computational advantage for the server, so to obtain a token of value $v = w + u$ the server should be forced to do an average of 2^{w+u} trial compuations. The $b = h(s\|b'), a = h(s\|b\|x), b \overset{\text{left}}{=}_w a \overset{\text{right}}{=}_u b'$ construct ensures this. Changing any of the values s, b' or x (b is changed by changing b') in any of the corresponding expressions for calculating b, a or b' has side effects on at minimum one of the other expressions. This property prevents the server finding a pair of collisions and then varying an independent parameter in the final expression calculation to match, which would admit a $2^w + 2^u < 2^{w+u}$ work factor attack.
- The parameters s, b' and b are all hashed before or during being used to find a collision. This ensures that similar service-names (service-names which are pre-fixes of other service-names), and similar blinding values don't offer the attacker any reduction in work factor by combining his attempts to create tokens for multiple services.
- Note the hash function should not admit any significant advantage in computing the hash operation by chosing a stride to match the hash operation internal structure (ie rather than chosing $x_0 \in_R \{0,1\}^\star$ and $x_{i+1} = x_i + 1$, use recurrence relation $x_{i+1} = f(x_i)$ with $f()$ chosen to reuse internal computation steps. Similarly no significant advantage should be achieved by computing a batch of candidate x_i values in parallel.

References

[1] BACK, A. Hashcash. http://www.cypherspace.org/ adam/hashcash/, 1997.

[2] DWORK, C., AND NAOR, M. Pricing via processing or combatting junk mail. In *Proceedings of Crypto* (1992).

[3] FRANKLIN, M., AND MALKHI, D. Auditable metering with lightweight security. In *Financial Cryptography* (1997), pp. 151–160.

[4] JUELS, A., AND BRAINARD, J. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Network and Distributed System Security Symposium* (1999).

[5] RIVEST, R., SHAMIR, A., AND WAGNER, D. Time–lock puzzles and timed–release crypto. In *Proceedings of Crypto* (1996).