# Lastregelung von Web Services mittels Proof-of-Work Funktionen

Sebastian Golze, Gero Mühl, Torben Weis, Michael C. Jäger

Kommunikations- und Betriebssysteme
Technische Universität Berlin
Fakultät IV, Sekretariat FR 6-3
Franklinstr. 28/29, D-10587 Berlin
{golze, gmuehl, weis, mcj}@ivs.tu-berlin.de

**Abstract:** Kostenlos frei angebotene Web Services sind anfällig für Überlastsituationen, die beispielsweise durch eine zu große Anzahl von Clients oder durch zu häufig anfragende Clients verursacht werden können. Auch Denial-of-Service Angriffe durch bösartige Clients können Überlastsituationen verursachen. Eine Lösung wäre, den betroffenen Web Service kostenpflichtig zu machen. Dies scheitert jedoch in der Praxis meist an der umständlichen Erhebung.
In diesem Artikel erläutern wir, wie mittels Proof-of-Work Funktionen eine Lastregelung von Web Services realisiert werden kann. Proof-of-Work Funktionen ermöglichen Clients nachzuweisen, dass sie eine gewisse Menge von Rechenleistung oder anderen Ressourcen erbracht haben. Durch die Regelung wird sichergestellt, dass alle Clients weiterhin Zugriff haben, aber die Zugriffe von zu häufig nachfragenden Clients eingeschränkt werden.

## 1 Einführung

Im Fall von übermäßigen Zugriffen oder sogar Denial-of-Service Angriffen müssen Web Services mit Überlastsituationen umgehen können, um ihre Funktion sicherstellen zu können. Die klassische Vorgehensweise in solchen Fällen ist, nur eine gewisse Anzahl von Anfragen zu bedienen, während der Rest abgewiesen wird. Der Nachteil dieser Technik ist, dass nicht zwischen legitimen Zugriffen und z.B. Denial-of-Service Zugriffen unterschieden wird. Sinnvollerweise sollten aber diejenigen Clients bedient werden, welche das höchste Interesse am zur Verfügung gestellten Dienst haben. Dieses Verteilungsproblem beschränkt sich nicht nur auf die Informatik, sondern existiert in vielen Bereichen, in denen knappe Ressourcen zugeteilt werden müssen. So erfüllt z.B. eine längere Warteschlange vor einem Schalter unter anderem diesen Zweck. Nur die wirklich interessierten Kunden werden in diesem Fall bereit sein, auf den Service zu warten. In vielen anderen Bereichen erfolgt die Steuerung über den Preis des Services. So variieren Strom- und Telefonanbieter ihre Preise je nachdem, wie stark ein Service zu einem gewissen Zeitpunkt beansprucht wird, um die Nachfrage zu steuern.

Eine nahe liegende Möglichkeit, die Nachfrage nach einem Web Service zu beeinflussen, wäre Gebühren für die Benutzung des Dienstes zu verlangen und die Höhe der Gebühren entsprechend der Nachfrage zu variieren. In der Praxis treten bei dieser Vorgehensweise aber häufig Umsetzungsschwierigkeiten auf. Obwohl Micropayment-Systeme

immer wieder angekündigt werden, hat sich noch kein System auf dem Markt etablieren können. Es wäre natürlich auch denkbar, die Zugriffe direkt zu berechnen, aber der administrative Aufwand wäre immens und daher kaum lohnend. Darüber hinaus würde dies eine ganze Reihe schwieriger Fragen bezüglich der Anonymität der Surfer und der Sicherheit des Bezahlvorgangs aufwerfen. Eine monatliche Pauschale wäre zwar unter Umständen umsetzbar, hätte aber, nachdem der Kunde gewonnen wurde, keinen ausreichenden Lenkungseffekt.

Weil es uns auch nicht primär darum geht, durch die Laststeuerung Einnahmen zu erzielen, bietet sich eine andere Alternative an: Bezahlen mit Rechnerressourcen. Dwork und Naor [DN92] schlagen vor, Spam Emails durch mittelschwere kryptografische Rätsel einzudämmen. Während die Lösung des Rätsels einen (in etwa) vorhersagbaren Aufwand erfordert, ist die Überprüfung der Lösung mit minimalem Aufwand möglich. Die Rätsel werden auch als Proof-of-Work Funktionen bezeichnet. In diesem Artikel beschreiben wir, wie Proof-of-Work Funktionen genutzt werden können, um die Last eines Web Services zu regeln. Hierdurch wird die zur Verfügung stehende Rechenkapazität näherungsweise fair auf alle Clients verteilt.

Die weitere Struktur dieses Artikels gliedert sich wie folgt: In Abschnitt 2 erläutern wir verschiedene Arten von Proof-of-Work Funktionen. Abschnitt 3 beschreibt die Regelung der Last durch einen einfachen Adaptionsalgorithmus. Anschließend beschreiben wir die prototypische Implementierung in Abschnitt 4 und besprechen verwandte Arbeiten in Abschnitt 5.

## 2 Proof-of-Work Funktionen

Es können drei aktuell gebräuchliche Formen von Proof-of-Work Funktionen unterschieden werden: Rechenzeit- und Speicherbasierte Funktionen sowie Touring Tests. Im Folgenden werden diese Funktionen kurz vorgestellt.

### 2.1 Rechenzeitbasierte Proof-of-Work Funktionen

Diese Funktionen basieren auf CPU-lastigen Algorithmen. Durch die beschränkte Leistungsfähigkeit der Hardware muss eine gewisse Rechenzeit aufgewendet werden, um eine solche Funktion zu lösen. Der Vorteil dieser Familie von Funktionen gegenüber anderen Verfahren ist, dass sie allgemein relativ leicht implementiert werden können. Es existieren Funktionen mit festem Berechnungsaufwand, wie z.B. Wurzelberechnungen, sowie Funktionen mit probabilistischem Aufwand, wie z.B. Hashkollisionen, welche die aktuell gängigste rechenzeitbasierte Proof-of-Work Funktion sind.

Bei Hashkollisionen bekommt der Client einen vorgegebenen String, hängt an diesen einen zweiten selbst gewählten String an und berechnet eine vorgegebene Hashfunktion auf dem Gesamtstring. Der Client probiert solange verschiedene Strings, bis das Ergebnis der Hashfunktion mit einer vorgegebenen Anzahl von Nullbits beginnt. Je mehr Nullbits gefordert werden, desto mehr Strings muss der Client durchprobieren, um einen

passenden String zu finden. Das Ergebnis dieses Verfahrens kann hingegen sehr leicht überprüft werden. Die Hashfunktion muss nur ein einziges Mal auf den vom Client gelieferten Gesamtstring angewendet werden und die Anzahl der Nullbits überprüft werden. Der Nachteil der rechenzeitbasierten Proof-of-Work Funktionen besteht darin, dass die Rechenleistung aktuell verwendeter Hardware sehr stark variiert, so dass legitime Clients mit älteren Prozessoren unter Umständen übermäßig benachteiligt werden.

### 2.2 Speicherbasierte Proof-of-Work Funktionen

Während bei der CPU Leistung immer wieder große Fortschritte gelungen sind, ist die Zugriffsgeschwindigkeit auf den Hauptspeicher deutlich langsamer gewachsen. Deswegen werden in [AB03] und [DGN03] Funktionen entwickelt, deren effizienteste Berechnung durch eine große Anzahl von Speicherzugriffen erfolgt. Die Datenmenge im Speicher muss dabei so groß gewählt werden, dass eine umfassende Nutzung des um ein Vielfaches schnelleren CPU Cache ausgeschlossen wird.

Die Nachteile der bekannten speicherbasierten Funktionen bestehen darin, dass eine Art Lookup Table von erheblicher Größe (z.B. 32MB) vor der Berechnung der Funktion in den Speicher des Clients übertragen werden muss. Erfolgt dies über das Internet, kann es zu erheblichen Verzögerungen kommen. Darüber hinaus ist die Implementierung dieser Funktionen nicht so einfach, wie die der meisten rechenzeitbasierten Funktionen, und es existieren weniger Forschungsergebnisse zur Sicherheit der Funktion. Sollte sich ein Weg finden, die Berechnungen abzukürzen, wären die Funktionen in der aktuellen Form nicht mehr brauchbar. Dies wäre auch der Fall, wenn zukünftige CPUs mit übermäßig großen Caches ausgestattet würden, so dass dieser die kompletten Lookup Tabellen aufnehmen könnte.

### 2.3 Touring Test-basierte Verfahren

Ein automatisierter Touring Test ist ein Verfahren, das Menschen und Maschinen unterscheiden kann. Diese Tests sind auch unter der Bezeichnung CAPTCHA (Completely Automated Public Turing Test to Tell Computers and Humans Apart) bekannt. Um Menschen von Maschinen zu unterschieden, wird dem Client eine Aufgabe gestellt, die gewöhnlich Menschen ohne große Probleme lösen können, während Computer hierzu nicht in der Lage sind. Häufig werden Bilder mit stark verzerrter Schrift erzeugt und der Benutzer aufgefordert, den enthaltenen Text zu extrahieren. [ABL04] und [ABHL03] geben eine Einführung in diese Tests. Große Webmail Provider benutzen z.B. solche Tests um zu verhindern, dass große Mengen von Benutzerkonten durch entsprechende Web Robots angelegt werden.

Der Nachteil dieser Tests, bei denen im Sinn des Proof–of-Work mit der Aufmerksamkeit des Benutzers bezahlt wird, besteht darin, dass sie fast immer auf offenen Problemen der künstlichen Intelligenz basieren. So ist es in der Vergangenheit gelungen, durch verbesserte Verfahren einige CAPTCHA Tests maschinell zu lösen. Dies führt zwar dazu, dass die künstliche Intelligenz als Wissenschaft voran gebracht wird, aber als Proof-of-Work Funktion sind diese Aufgabenstellungen nicht mehr ohne weiteres nutz-

bar. Darüber hinaus können diese Aufgaben nicht im Hintergrund unsichtbar für den Benutzer berechnet werden und lösen bei vielen Benutzern Befremden aus. Ein weiterer Nachteil ist, dass die Erstellung der benötigten Rätsel auch mit einem im Vergleich zu den anderen Verfahren relativ hohen Aufwand für den Server verbunden ist, so dass sich die Anwendung nur beim Zugriff auf größere Ressourcen lohnt, da sonst durch den Proof-of-Work mehr Serverleistung verloren geht, als durch die Lastregelung gewonnen werden kann. Aber insbesondere im Fall von mobilen Endgeräten, die Aufgrund Ihrer Hardware unter Umständen kein nennenswertes CPU- oder speicherbasiertes Proof-of-Work erbringen können, stellen Touring Tests eine interessante Alternative dar.

## 3. Lastregelung mittels Proof-of-Work Funktion

Im Unterschied zu den bei der Spam-Vermeidung eingesetzten Proof-of-Work Funktionen, bei denen die Schwierigkeit der Berechnung fest steht und nur dem technischen Fortschritt angepasst wird, muss die Proof-of-Work Funktion im Fall der Web Service-Lastregelung zeitnah an die aktuelle Last angepasst werden. Um diese Regelung entwerfen zu können, muss unter anderem geklärt werden, ob der Preis pro Zugriff konstant berechnet wird oder nicht. Im Folgenden werden kurz die Vor- und Nachteile der beiden Verfahren erläutert.

### 3.1 Konstante Preisberechnung

Bei der konstanten Preisberechnung wird immer derselbe Preis je Zugriff erhoben, unabhängig davon, wie viele Zugriffe vom jeweiligen Client eintreffen. Dies hat den Vorteil, dass keine Verzerrungen auftreten, sollte sich ein Client hinter mehreren IP-Adressen als Rechnergruppe tarnen oder falls mehrere Clients hinter einer Adresse arbeiten. Wir schlagen für die Regelung der Proof-of-Work Funktion folgende Anpassung vor:

$$P_n = (R/C) \cdot P_c$$

$P_n$: Neue Proof-of-Work Anforderung
$R$: Anzahl der Zugriffsversuche
$C$: Maximal angestrebte Anzahl der Zugriffe
$P_c$: Aktuelle Proof-of-Work Anforderung (Nicht Null)

Da die Berechnung eines Proof-of-Work eine gewisse Zeit dauert und begonnene Berechnungen noch eine gewisse Zeit akzeptiert werden, darf die Anpassung nicht in kürzeren Intervallen erfolgen, als die durchschnittliche Berechnungsdauer des aktuellen Proof-of-Work beträgt. Ansonsten würde ein Übersteuern auftreten.

### 3.2 Exponentielle Preisberechnung

Im Geschäftsleben ist es üblich, Kunden bei der Abnahme größerer Mengen Rabatte einzuräumen. Dies ist möglich, weil die Gemeinkosten auf die größere Menge umgelegt werden können und daher die Hoffnung besteht, dem Kunden mehr zu verkaufen, als er bei einem konstanten Preis kaufen würde. Unser Ziel ist es aber, die Zugriffe zu be-

schränken statt weitere Zugriffe zu generieren. Deswegen schlagen wir genau das Gegenteil des üblichen Mengenrabatts vor. Je mehr Zugriffe ein Client durchführen will, desto höher wird der Preis pro Zugriff. Ein Problem hierbei ist, dass ein Client dem Web Service mehrere „virtuelle“ Clients vorgaukeln könnte, um dem exponentiellen Aufwand zu entgehen, zum Beispiel durch die Verwendung mehrerer IP-Adressen. Im schlimmsten Fall, wenn ein Client so viele „virtuelle“ Clients vorspielt, wie er Zugriffe in einem Regelungsintervall abgibt, so entsteht ihm nur ein konstanter Aufwand. Daher ist die exponentielle Preisberechnung in dieser Beziehung nie schlechter als die konstante. Ein weiteres Problem sind z.B. Proxy oder Network Address Translation Server. Diese würden ungerechtfertigt mit zu aufwändigen Proof-of-Work Funktionen belegt werden, weil sie dem Web Service wie ein einzelner Client erscheinen.

Die folgende Formel definiert den Preis im Fall der exponentiellen Preisberechnung:

$$P_n = k^n \cdot P$$

$P_n$: Proof-of-Work für den $n$-ten Zugriff
$k$: Konstante zur Definition der Preiskurve
$n$: Bisherige Anzahl der Zugriffe im Intervall
$P$: Basis Proof-of-Work

Die Nachregelung der exponentiellen Proof-of-Work Funktion ist nicht ganz so einfach zu realisieren wie die der konstanten, weil die Auswirkung eines sich ändernden Basis Proof-of-Work sehr stark von der Verteilung der Zugriffe auf die Clients abhängig ist. Durch die unterschiedlich schweren Funktionen kann darüber hinaus kein einfaches und sinnvolles Intervall angeben werden, nach dem eine komplette Nachregelung erfolgen sollte. Deswegen schlagen wir vor, die Funktion im regeltechnischen Sinn als einschleifigen Regelkreis zu betrachten. Die Stellgröße ist die Basis Proof-of-Work Funktion und die Ausgangsgröße ist die Last. Zur Regelung schlagen wir folgenden einfachen Algorithmus vor: Solange die Serverlast über dem Zielwert liegt wird die Schwierigkeit des Basis Proof-of-Work verdoppelt. Sollte die Last geringer werden, werden die Anforderungen halbiert. Zwischen zwei aufeinanderfolgenden Anwendungen der Funktion muss, um eine Übersteuern zu vermeiden, wieder eine gewisse Zeit gewartet werden. Diese Zeit sollte an die aktuelle Proof-of-Work Funktion angepasst werden. Diese grobe Regelung kann ergänzt werden, indem bei einem Wert, der nahe am Zielwert liegt, die Schwierigkeit des Basis Proof-of-Work nicht verdoppelt bzw. halbiert, sondern linear nachführt, um ein gleichmäßigeres Verhalten zu erhalten.

## 4. Prototypische Implementierung

Um unseren Ansatz zu testen, haben wir einen entsprechenden Prototyp realisiert. Hierfür haben wir die CPU-basierten Hashkollisionen als Proof-of-Work Funktion ausgewählt. Adam Black stellt unter [B04] entsprechende Bibliotheken zur Verfügung. Vor der eigentlichen Implementierung haben wir diese Funktion einigen Tests unterzogen. Die entsprechende Bibliothek liegt in verschiedenen Programmiersprachen vor. Unsere Messungen haben ergeben, dass die Java Version mit einer Hot Spot Java Virtual Maschine im Schnitt nur ca. 25% der Leistung der nativen C Implementierung erreicht. Das

bedeutet, dass in jedem realen Einsatzszenario, die native Bibliothek zum Einsatz kommen sollte, weil sonst legitime Clients gegenüber böswilligen Clients, die höchst wahrscheinlich auf die native Bibliothek zurückgreifen werden, unnötig benachteiligt wären.

Statistisch gesehen, verdoppelt sich der durchschnittliche Aufwand, um eine passende Hashkollision zu finden, wenn die Anzahl der geforderten Bitkollisionen um eins erhöht wird. Der Berechnungsaufwand von Hashkollisionen ist allerdings nur probabilistisch vorhersagbar. Dies liegt daran, dass je nachdem, wo der Client beginnt, den Suchraum zu durchsuchen, es sehr schnell gehen oder auch relativ lange dauern kann, bis eine Kollision entdeckt wird. Deswegen sehen wir in unserer Implementierung die Möglichkeit vor, von den Clients nicht die Lösung eines großen Proof-of-Work zu verlangen, sondern mehrere leichtere, die statistisch gesehen etwa so lange dauern, wie ein großes. Die Anwendung mehrerer leichter Rätsel hat den Vorteil, dass der zeitliche Aufwand für die gesamte Berechnung weniger streut. Tabelle 1 zeigt Ergebnisse entsprechender Messungen, die diesen Effekt verdeutlichen. Dabei sollte das Proof-of-Work aber nicht zu weit aufgespaltet werden, weil für jedes einzelne Proof-of-Work ein Ergebnis übertragen werden muss. Dies kostet mehr Bandbreite.

| Nullbits | 1x19bit | 4x17bit | 16x15bit | 64x13bit |
|---|---|---|---|---|
| ∅ Dauer | 5412 ms | 6111 ms | 5905 ms | 5396 ms |
| Std. Abweichung | 5118 ms | 3007 ms | 1814 ms | 965 ms |

Tabelle 1: Gemessene Berechnungsdauer von Hashkollisionen (jeweils 50 Messungen)

Unser Prototyp setzt direkt im http Protokoll an. Solange der Server unter einer definierten Lastobergrenze von z.B. 80% arbeitet, läuft das http Protokoll ganz normal. Sollte dieser Wert überschritten werden, greift die Proof-of-Work Lastregelung. Dazu antwortet der Server nicht wie gewöhnlich mit einem Code 200 sondern mit dem Code 402 "Payment Required". Darüber hinaus fügt der Server eine Zeile im folgenden Format in den Antwort-Header ein: "X-POW: hashbits=20&hashcount=4&valid=30"

Diese Zeile gibt im URL Format mit dem Parameter hashbits an, wie viele Nullbits gefordert werden, mit hashcount, wie viele dieser Kollisionen zu liefern sind, und mit valid, wie viele Sekunden diese Parameter gültig sind. Der Client berechnet die geforderte Anzahl Hashkollisionen mit der angeforderten URL und dem Datum als Basis String. Danach startet er einen neuen http Zugriff, wobei er dem Header die Zeile "X-POW:" hinzufügt und die ermittelten Kollisionen mit Semikolon getrennt hier angibt. Der Server überprüft diese und liefert, sollten sie korrekt sein, die geforderten Daten. Der Einfachheit halber arbeiten wir mit konstantem Pricing, weil sonst die Zugriffe der einzelnen Clients gezählt werden müssten und dies dem Client so kommuniziert werden müsste, dass dieser bei mehreren parallel laufenden Zugriffen immer mit den richtigen Parametern arbeitet.

Wir haben Messungen mit einem Server, einem bösartigen Client (700 MHz CPU), der eine Denial-of-Service Attacke durchführt, und einem legitimen Client (500 MHz CPU) durchgeführt. Der bösartige Client startet so viele Zugriffe, wie es ihm mit der zur Verfügung stehenden Rechenleistung möglich ist, während der legitime Client nur einen Zugriff alle zwei Sekunden absendet. Ohne die Verwendung eines Proof-of-Work wird

der Server irgendwann überlastet und es wird nur noch eine gewisse Quote von Zugriffen bearbeitet. Dies führt dazu, dass auch Zugriffe des legitimen Clients nicht mehr bearbeitet werden. Wenn ein Proof-of-Work verwendet wird, muss der legitime Client zwar in unserem Beispielszenario gemittelte 16% seiner Rechenzeit für die Errechnung der Proof-of-Work Funktionen aufwenden. Dafür wird aber keiner seiner Zugriffe abgewiesen. Ein Szenario mit mehreren Clients unterscheidet sich nur unwesentlich vom untersuchten Fall, weil durch die Proof-of-Work Funktion nur das Verhältnis zwischen den verfügbaren Rechenressourcen der legitimen und der bösartigen Clients entscheidend ist, aber nicht deren Anzahl.

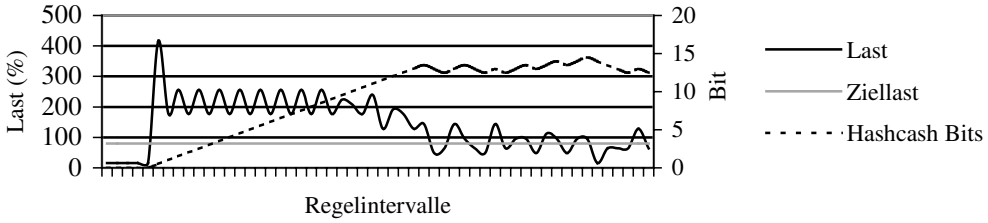


Abb. 1: Lastverhalten des Web Service beim Zuschalten eines bösartigen Clients

Abb. 1 zeigt das Lastverhalten des Web Service beim Zuschalten eines bösartigen Clients. Es ist zu erkennen, wie die Anforderungen an das Proof-of-Work so lange erhöht werden, bis sich die Last einpendelt. Hierbei ist zu beachten, dass die Rechenzeit des Clients für das Proof-of-Work exponentiell steigt, wenn die Anzahl der geforderten Bits linear steigt.

## 5 Verwandte Arbeiten

Neben den bereits genannten Arbeiten möchten wir an dieser Stelle auf weitere verwandte Arbeiten hinweisen. Jakobsson und Juels [JJ99] untersuchen die Frage, wie die für das Proof-of-Work aufgewendete Rechenleistung über den Ernsthaftigkeitsnachweis hinaus zur Lösung anderer Aufgaben verwendet werden kann. Diese Fragestellung ist wichtig, weil ein oft gegen Proof-of-Work angeführtes Argument ist, dass es nicht zu verantworten sei, wertvolle Rechenleistung zu „verschwenden“. Juels und Brainard [JB99] beschreiben einen Ansatz, mit dem sich *connection depletion* Angriffe auf das TCP- und das SSL-Protokoll abwehren lassen. Die vorgeschlagene Lösung zielt aber nur auf bösartige Angriffe ab. Die Autoren gehen nicht auf eine mögliche Laststeuerung ein. Goldschlag und Stubblebine [GS98] haben ein Verfahren entwickelt, wie sich mittels Proof-of-Work Funktionen (die sie "Delaying Functions" nennen) eine Lotterie realisieren lässt, deren Gewinner durch jeden Teilnehmer berechnet werden kann. Allerdings dauert die Berechnung der Gewinner durch die Verwendung von Proof-of-Work Funktionen länger, als die Lotterie dauert. Dadurch können Teilnehmer zwar das Ergebnis im Nachhinein überprüfen, aber sich nicht vor dem Ende der Lotterie einen Vorteil verschaffen. Die Arbeiten von Juels und Brainard und von Goldschlag und Stubblebine sowie unser vorliegender Artikel zeigen mögliche Anwendungsfälle von Proof-of-Work Funktionen außerhalb der Vermeidung von Spam-E-Mails.

## 6 Zusammenfassung

In diesem Artikel haben wir beschrieben, wie Proof-of-Work Funktionen zur Lastregelung eines Web Services verwendet werden können. Wir haben verschiedene Möglichkeiten der Preisberechnung vorgestellt und deren Auswirkungen diskutiert. Den vorgestellten Ansatz haben wir prototypisch implementiert und entsprechende Messungen präsentiert. Diese zeigen die Anwendbarkeit unseres Ansatzes. Zukünftige Arbeiten in diesem Themenumfeld werden unter anderem eine verbesserte Steuerung der Proof-of-Work Schwierigkeit sowie den Einsatz anderer Funktionen als der Hashcash Funktion beinhalten. Darüber hinaus beabsichtigen wir zu untersuchen, in wiefern der Proof-of-Work Ansatz durch das Erschleichen von Rechenleistung (z.B. durch Internetwürmer) unterlaufen werden könnte.

## Literaturverzeichnis


[ABL04] AHN Luis von, BLUM Manuel, LANGFORD John: *Telling Humans and Computers Apart (Automatically)* In: Communications of the ACM, Volume 47, Issue 2, Seiten 56 – 60, 2004

[AB03] ABADI Martin, BURROWS Mike, MANASSE Mark, WOBBER Ted: *Moderately Hard, Memory-bound Functions*, In: Proceedings of the 10th Annual Network and Distributed System Security Symposium, 2003
*http://www.hashcash.org/papers/memory-bound-ndss.pdf*

[ABHL03] AHN Lous von, BLUM Manuel, HOPPER Nicholas J., LANGFORD John: *CAPTCHA: Using Hard AI Problems For* Security In: Advances in Cryptology, Eurocrypt 2003

[B04] BACK Adam: Hashcash - A Denial-of-Service Counter-Measure http://www.hashcash.org/

[DGN03] DWORK Cynthia, GOLDBERG Andrew, NAOR Moni: *On Memory-Bound Functions for Fighting Spam*, Microsoft Research In: Proceedings of the 23rd Annual International Cryptology Conference (CRYPTO 2003), Seiten 426-444, Springer 2003 http://research.microsoft.com/research/sv/PennyBlack/demo/lbdgn.pdf

[DN92] DWORK Cynthia, NAOR Moni: *Pricing via Processing or Combatting Junk Mail* In: LNCS Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology, Seiten 139 - 147, 1992

[GS98] GOLDSCHLAG David M., STUBBLEBINE Stuart G.: *Publicly Verifiable Lotteries: Applications of Delaying Functions* In: LNCS 1465 Seiten 214 – 226, 1998

[JB99] JUELS Ari, BRAINARD John: *Client Puzzles: A Cryptographic Defense Against Connection Depletion Attacks* In: Proceedings of NDSS '99 (Networks and Distributed Security Systems), Seiten 151-165, 1999

[JJ99] JAKOBSSON Markus, JUELS Ari: *Proofs of Work and Bread Pudding Protocols* In: B. Communications and Multimedia Security, Seiten 258-272, Kluwer Academic Publishers, 1999