KTH Computer Science
and Communication

Proof of Work

En studie av proof-of-work som skydd mot överbelastningsattacker

FREDRIK HENRIQUES
SIGTUNAGATAN 12, 113 22 STOCKHOLM
070-716 84 93, HENRIQ@KTH.SE
NIKLAS NORDMARK
METARGATAN 28, 116 66 STOCKHOLM
070-798 45 12, NIKAN@KTH.SE

Kandidatexamenrapport vid NADA, kurs DD143X
Handledare: Mikael Goldmann
Examinator: Mads Dam

Referat

Denna uppsats ämnar förklara protokollet proof-of-work samt utvärdera dess förmåga att skydda en nätverksresurs från att göras otillgänglig av en överbelastningsattack. Vi vill genom tester på en implementation av protokollet svara på vilken form av pussel som ger en rättvis arbetsfördelning för klienterna. Resultatet visar att proof-of-work kan skydda en nätverksresurs genom att begränsa antalet anrop som varje klient kan göra per tidsenhet. Vi ser även att en implementation med delpussel ger en mindre variation i beräkningstider och är därför att föredra framför en svårare implementation utan delpussel. Klienter med mindre kraftfulla datorer missgynnas dock fortfarande i förhållande till klienter med kraftfullare maskinvara.

Abstract

Proof of Work

The purpose of this essay is to explain the proof of work protocol and to evaluate its ability to protect a network resource from denial of service attacks. By implementing a proof of work protocol we aim to evaluate what kind of puzzles give an even workload to clients. The results prove that proof of work can be used to protect a network resource by limiting the amount of calls a client can perform in a certain time. It is also evident that when subpuzzles are used there is less variation of how long it takes to solve a puzzle, therefore subpuzzles are to be prefered over implementations without subpuzzles. Even though variations in calculation times are reduced, clients with low performance hardware will still have a disadvantage in relation to clients with more powerful hardware.

Redogörelse av samarbete

Vid arbetets början så arbetade Niklas med att ta fram material och skriva bakgrund till uppsatsämnet medan Fredrik började programmera på prototypen. Vi har beslutat hur vi ska besvara vår frågeställning, genomfört simuleringar samt skrivit analysen och slutsatserna tillsammans. All text har korrigerats och diskuterats gemensamt.

Innehåll

Kapitel 1

Introduktion

1.1 Introduktion

Användandet av nätverksbaserade tjänster har ökat explosionsartat i och med internets breda genomslag under de senaste två decennierna. Det har redan från början varit ett stort problem att illasinnade användare kunnat använda sig av olika typer av överbelastningsattacker för att begränsa åtkomligheten av nätverkstjänster [1]. Detta kan exempelvis ske genom upprepade anrop till en nätverkstjänst som i strömmen av illasinnade anrop inte kan behandla de anrop som kommer från normala användare. En annan form av överbelastning är den stora mängden skräppost som dagligen skickas till mailservrar. Detta kan orsaka problem för både användare och för mailservrar i arbetet att avgöra vad som är läsvärt för användaren och vad som endast innehåller reklam eller virus.

1.2 Bakgrund

Redan 1992 introducerades begreppet proof-of-work som en metod för att garantera tillgänglighet för nätverkstjänster i allmänhet och med huvudsyftet att bekämpa skräppost [2]. Grundidén var att belägga anrop till en server med en kostnad, dock ej en monetär sådan. Kostnaden skulle bestå av att en viss mängd arbete utfördes av klienten innan servern utförde det efterfrågade arbetet. Klienten skickar ett bevis för utfört arbete, proof-of-work, till servern som validerar att klienten verkligen utfört arbetet innan den ges tillgång till resursen. En av de mer kända implementationerna är *hashcash* [3].

Gemensamt för de olika implementationerna är att arbetet som krävs av klienten för att få använda servertjänsten är relativt svårt att utföra medan verifikationen som utförs av servern är betydligt enklare. Beräkningen får dock inte vara så svår att en normal användare påverkas i för stor utsträckning, vilket medför ett problem då beräkningshastigheten kan variera kraftigt mellan olika klienter. För att minska skillnaderna i beräkningstid mellan olika klienter har versioner av protokollet med

beräkningar som är beroende av internminnet snarare än processorn prövats. Detta eftersom internminnens hastighet oftast skiljer sig mindre än processorhastigheten mellan olika klienter [4]. Det finns även versioner där hela arbetet inte utförs av klienten utan delvis av en annan klient som servern väljer ut slumpmässigt vilket resulterar i att en stark egen dator inte ger en stor fördel i beräkningen av proof-of-work [1].

De flesta av dessa modeller har haft som huvudsakliga mål att bekämpa skräppost vilket proof-of-work visat sig vara otillräckligt för. Klienternas beräkningar har visat sig behöva vara väldigt tidskrävande för att det inte skulle vara ekonomiskt försvarbart för en spammare att utföra dem. Spammare behöver dessutom ofta inte göra beräkningarna från egna datorer, utan använder sig ofta av bot-nät. Om beräkningskostnaden höjdes till en nivå som gjorde det ekonomiskt oförsvarbart att skicka spam så skulle normala användare inte kunna skicka mail inom rimlig tid [5].

Proof-of-work kan fortfarande användas för att begränsa åtkomsten till nätverksresurser för att på så sätt garantera att de aldrig helt överbelastas. Ett konkret exempel på ett användningsområde är i en nätverkstjänst bestående av ett flertal webbservrar som delar på samma databasserver. Genom att tvinga användare som kontaktar webbservern att utföra ett proof-of-work innan webbservern skickar en resurskrävande förfrågning, exempelvis en sökning, till databasservern vill man garantera att även om en webbserver är under attack så ska databasservern inte bli överbelastad och därmed otillgänglig för de andra webbservrarna.

1.3 Frågeställningar och syfte

I den här uppsatsen kommer vi att analysera en implementation av algoritmen ”hashberäkning med delpussel” för proof-of-work. Implementationen kommer att användas för att avgöra:

- Hur effektivt proof-of-work kan skydda en bakomliggande resurs.
- Hur legitima klienter påverkas av proof-of-work.
- Vilka styrkor och svagheter som finns med protokollet hashberäkning samt hur dessa ändras när man introducerar delpussel.

Syftet med uppsatsen är att avgöra hur bra skydd proof-of-work ger samt att utvärdera ifall det är användbart i praktiken eller inte.

1.4 Uppsatsens uppbyggnad

Kapitel 2 beskriver proof-of-work protokollet mer ingående. 2.1 beskriver arbetsflödet medan 2.2 ger en inblick i två olika val av pusselalgoritm. I kapitel 3 beskrivs

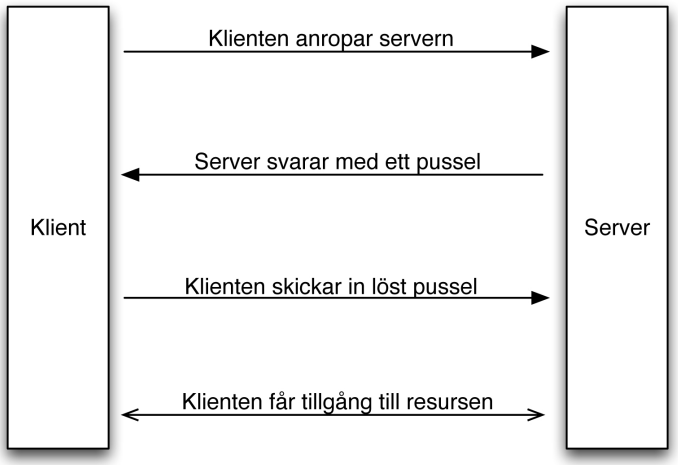
den implementation vi skrivit för att utvärdera proof-of-work tillsammans med en fördjupning i det pussel vi valt att implementera. Kapitel 4 presenterar och analyserar framräknade och uppmätta värden för olika svårighetsgrader av pussel i ett försök att finna ett pussel som ger en rättvis och mindre slumpmässig beräkningstid för olika klienter. Våra slutsatser presenteras i kapitel 5.

Kapitel 2

Proof of Work

2.1 Arbetsflöde

Arbetsflödet i de vanligaste implementationerna av proof-of-work inleds med att klienten skickar ett anrop till servern. Servern svarar med att skicka ett meddelande innehållande information om arbetet som krävs för att beviljas åtkomst till nätverkstjänsten. När klienten utfört det efterfrågade arbetet skickas ett svar tillbaka till servern för verifiering. Beroende på vad för nätverkstjänst som begränsas av servern kan det vara nödvändigt att anpassa svårighetsgraden på arbetet efter nuvarande belastning, eller till och med stänga av proof-of-work vid låg belastning. När servern verifierat att klienten löst pusslet ges tillgång till nätverkstjänsten.



Figur 2.1. Arbetsflöde vid proof-of-work

2.2 Pussel

Det arbete som en klient måste utföra för att beviljas tillträde till den begränsade nätverkstjänsten kallas pussel. Ett pussel kan utformas på många olika sätt med varierande svårighetsgrad men det finns ett antal regler som måste uppfyllas för att proof-of-work-protokollet ska fungera.

- Det måste vara relativt svårt för klienten att beräkna en lösning. Svårighetsgraden bör anpassas efter flera faktorer som exempelvis arbetsbördan för nätverksresursen per anrop, nuvarande belastning och klientens tålamod.
- Det måste vara relativt enkelt för servern att verifiera ett svar. Verifieringens svårighetsgrad måste vara låg för att förhindra att själva verifieringen blir så resurskrävande att den i sig orsakar en överbelastning.
- Det får inte vara möjligt för en klient att förberäkna lösningar. Om en illasinnad klient kan skapa lösningar i förväg och spara dessa i en lista kan nätverkstjänsten lätt överbelastas. För att motverka detta måste pusslet ha en tidsbegränsning och vara oförutsägbart.

2.2.1 Primfaktoriering

Faktorisering av produkter av stora primtal har använts i många olika tillfällen inom datorsäkerheten, bland annat vid kryptering med RSA [6, sid 637]. Datorer kan idag enkelt multiplicera stora tal samt med hög sannolikhet avgöra ifall ett stort tal är ett primtal [7, sid 890], men att faktorisera ett stort tal är komplicerat. Även om algoritmer som Pollar-Rho [7, sid 897] och Quadraticsieve [8] gjort att tidskomplexiteten för primtalfaktorisering av stora tal minskat så är det fortfarande ett relativt svårt arbete för datorer att faktorisera primtal [7, sid 896]. Dessa egenskaper gör primtalsfaktorisering lämpligt som pussel i proof-of-work.

Att använda sig av primtalsfaktorisering för att skapa pussel har testats vid flera tillfällen genom att servern efter att ha anropats av en klient väljer två primtal av lämplig storlek, multiplicera dem och skicka resultatet till klienten. Klientens uppgift är sedan att faktorisera produkten och returnera talen till servern [2].

Fördelar:

- Verifiering av pusslet är väldigt enkel för servern i förhållande till hur svår en lösning är att beräkna för klienten.

Nackdelar:

- Servern måste antingen ha en lista av primtal eller räkna ut nya primtal vid varje anrop. Om servern har en lista skulle en illasinnad klient efter tillräckligt många anrop kunna återskapa listan och därigenom slippa faktorisera talen.

Om servern räknar ut nya tal vid varje anrop ökar belastningen på servern för varje anrop.

2.2.2 Hashberäkning

En typ av pussel till proof-of-work bygger på hashberäkningsalgoritmer. Den går till så att servern först väljer en svårighetsgrad på det pussel som ska skickas. Servern skickar sedan en sträng p av lämplig storlek till klienten vars uppgift är att konkatenera p med ett värde x. Klienten använder sedan en hashfunktion h för att beräkna $Y = h(x, p)$. Om man använder hashalgoritmen MD5 så blir Y en 32 tecken lång hexadecimal talföljd, vilket innebär att den kan anta 16^{32} olika värden. När klienten hittat ett Y som inleds med minst lika många nollor som svårighetsgraden anger så är pusslet löst, annars testar klienten med ett nytt x tills den hittar ett som x som ger en lösning [9].

$$h(x, p) = Y = \underbrace{0 \ldots 0}_{d\,nollor} + y < d, 31 >$$

Klienten skickar sedan x till servern som enkelt kan verifiera lösningen genom att beräkna $h(x, p)$ och kontrollera antal inledande nollor.

Fördelar:

- Verifiering av pusslet är väldigt enkel för servern i förhållande till hur svårt det är att räkna ut för klienten.

Nackdelar:

- Mängden arbete som måste utföras är fördelad över ett stort spann. Detta innebär att vissa klienter kan lösa sin uppgift väldigt snabbt medan andra måste vänta väldigt länge innan de har löst uppgiften, trots att svårighetsgraden är densamma.
- Pusslet begränsas av processorn i datorn, därför får en snabb klient vänta mycket kortare än en långsam klient.
- Pusslets svårighetsgrader är mycket ojämna, varje nivå blir sexton gånger svårare än den förra eftersom endast ett av sexton möjliga värden godkänns. Detta gör att det kan vara svårt att hitta en bra nivå.

Några av problemen med hashberäkning går att minska genom att introducera *delpussel* [10]. Detta innebär att man skickar ett antal pussel till varje klient att lösa istället för att endast skicka ett pussel av högre svårighetsgrad.

Kapitel 3

Metod

3.1 Implementation

För att besvara våra frågeställningar kommer vi att implementera en prototyp som använder hashberäkning med delpussel för att skydda en resurskrävande nätverkstjänst från överbelastningsattacker. Vi kommer att implementera proof-of-work i applikationslagret av nätverksstacken som ett steg man måste gå igenom innan man får tillgång till tjänsten; således kommer det inte skydda mot överbelastningsattacker som är inriktade mot den underliggande tekniken (t ex attacker mot tcp-protokollet) utan endast den skyddade tjänsten.

Prototypen kommer att bestå av två delar, en server och en klient, båda skrivna i Java. Koden finns tillgänglig för nedladdning på:
http://www.student.nada.kth.se/~henriq/POW/

3.1.1 Server

Servern är en mycket skalbar server som har designats från grunden upp för att klara av att behandla många samtidiga klienter. Servern har en trådpool bestående av trådar som kan behandla klienter som ansluter till servern. Om en ny klient ansluter så hämtar servern en ledig tråd till klienten, eller lägger klienten på kö om det inte finns några lediga trådar. Antalet trådar i trådpoolen anpassas dynamiskt till belastningen på servern för att man inte ska ha fler trådar än nödvändigt som tar minne samtidigt som man vill slippa det extra arbete som det innebär att ofta skapa nya trådar.

Vid bestämda tidsintervall så analyserar servern hur hög belastning den har och anpassar förutom antalet trådar även svårighetsgraden på proof-of-work pusslen som ges ut. Om servern har mycket belastning (vilket tyder på att den kan vara under attack) så kan den således börja ge ut svårare pussel för att minska belastningen, medans den vid låg belastning kan ge ut lätta pussel eller till och med stänga av proof-of-work. Stegen som används för att bestämma vad som är ”hög

belastning" sätts i variabler så att stegen kan anpassas efter hur många anrop per sekund nätverksresursen klarar av att hantera.

Datatyper

- **Seed** - Ett frö som används för att generera pussel. Fröet bör inte vara för litet och det bör bytas ut med jämna mellanrum för att förhindra att attackerare ska kunna förberäkna lösningar. I vårt fall har vi valt ett 128 tecken långt frö.
- **PUZZLE_TTL** – Den maximala tiden i sekunder ett pussel är giltigt från att klienten först kontaktade servern.
- **MAX_PUZZLES** – Ett id-nummer genereras för varje nytt anrop till servern, när id-numret når MAX_PUZZLES startas det om från noll. MAX_PUZZLES storlek måste vara större än det maximala antalet nya anrop som kan behandlas under PUZZLE_TTL för att klienter som tar lång tid på sig att lämna in sitt svar inte ska bli av med sitt id.
- **Puzzles** - En lista innehållandes MAX_PUZZLES antal element av typen pussel-metadata. Pussel-metadata innehåller en tid för att kontrollera om lösningen är inlämnad i tid samt svårighetsgraden på pusslet.
- **POWDifficulty** – En siffra som anger hur många inledande nollor som krävs för att en lösning ska godkännas. Svårighetsgraden sätts dynamiskt beroende på aktuell serverbelastning.

3.1.2 Klient

Klienten ansluter till servern med en eller flera trådar (flera trådar vid simulering). Varje tråd kommunicerar med servern som om den vore en egen klient, se figur 3.1. Om servern returnerar ett pussel så löser klienten pusslet och svarar med lösningen. Efter att klienten fått tillgång till tjänsten så börjar den om från början igen.

3.1.3 Algoritm

När en klient ansluter till servern så begär den att få tillgång till serverns resurser. Om servern har proof-of-work avstängt så ges den direkt tillgång. Om serverns har proof-of-work påslaget så genererar den ett nytt id som sparas tillsammans med tiden och svårighetsgraden för pusslet i listan Puzzles. Servern beräknar sedan med hjälp av hashalgoritmen MD5 ut pusslet $h(Seed, t)$. Beräkningen genererar ett 32 siffror långt hexadecimalt tal P som servern skickar till klienten tillsammans med id och svårighetsgrad d.

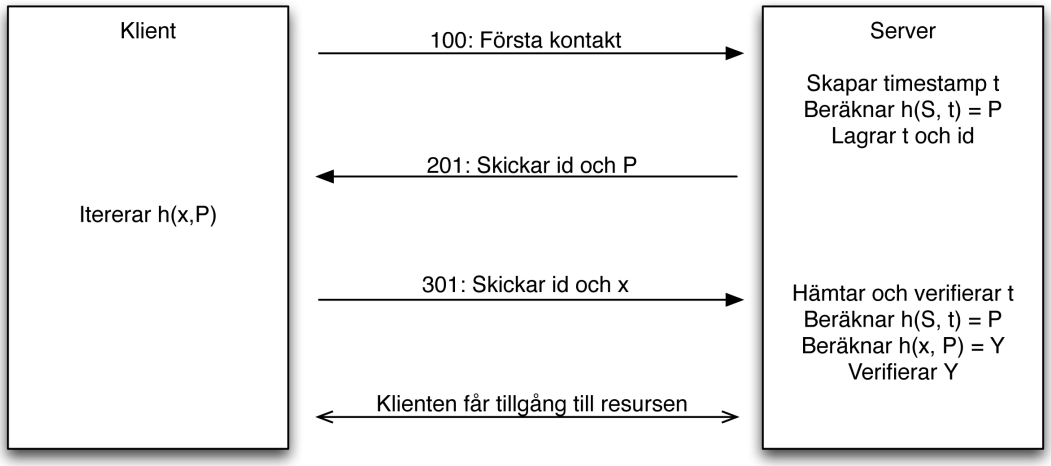
Klientens beräknar sedan en lösning genom att konkatenera en sträng x med P, beräkna $h(x, P)$ och kontrollera antalet inledande nollor. Proceduren upprepas med olika x till dess att hashvärdet har minst d inledande nollor. Klienten skickar sedan

ett nytt anrop till servern med lösningen x och id.

För att verifiera att klienten utfört sitt arbete och att det lämnats in i tid hämtas tiden och svårighetsgraden ur Puzzles med hjälp av id. Om skillnaden mellan den lagrade tiden och den nuvarande tiden inte är mindre än PUZZLE_TTL är pusslet giltigt och då beräknas återigen P från de lagrade värden $Seed$ och t. Sedan verifieras x genom att $h(x, P)$ beräknas och de inledande d siffrorna kontrolleras. Om lösningen är korrekt så ges klienten tillgång till resursen.

När delpussel används så berättar servern hur många lösningar den vill ha på samma problem. Klienten måste sedan svara med lika många unika lösningar som alla ska vara giltiga för att få tillgång till serverns resurser.

Eftersom klienten aldrig har tillgång till varken den exakta tiden t eller $Seed$ kan egna pussel ej konstrueras vilket förhindrar förberäkning av lösningar. $Seed$ bör bytas ut med jämna mellanrum eftersom det skulle vara möjligt för en klient, även om det vore svårt då den inte vet t, att räkna ut $Seed$ efter tillräckligt många anrop. Även om en klient skulle räkna ut $Seed$ så skulle det i praktiken vara mycket svårt att producera egna lösningar, eftersom servern hashar med t, tiden då anropet kom in i millisekunder, vilken är mycket svår för klienten att synkronisera med så stor precision.



Figur 3.1. En tydligare bild av arbetsflödet

3.1.4 Protokoll

Här följer en kort beskrivning av protokollet som används mellan klient och server.

Tabell 3.1. Beskrivning av protokollet som används vid klient-server kommunikation

Kod	**Betydelse**
100	Klienten hälsar på servern
101	Servern meddelar klienten att den har tillgång till resursen
201 XXXXX ID DELAR SVÅRIGHET	Servern skickar ett pussel XXXXX, id ID och begär DELAR antal lösningar (delpussel) med svårighet SVÅRIGHET
301 ID XXXX YYYY	Klienten skickar lösningarna XXXX och YYYY (vid två delpussel, antalet lösningar är detsamma som antalet server begärde) till pusslet med id ID.

3.2 Testfall

För att utvärdera effektiviteten av proof-of-work så kommer vi att utföra ett antal simuleringar med hjälp av prototyperna. Simuleringarna kommer att mäta tiden det tar för en klient att beräkna pussel vid olika svårighetsgrader samt med olika antal delpussel. Vi kommer att mäta dessa tider för olika processorer för att utvärdera hur stor skillnad i beräkningstid det tar för olika snabba klienter.

Kapitel 4

Analys och Resultat

4.1 Analys

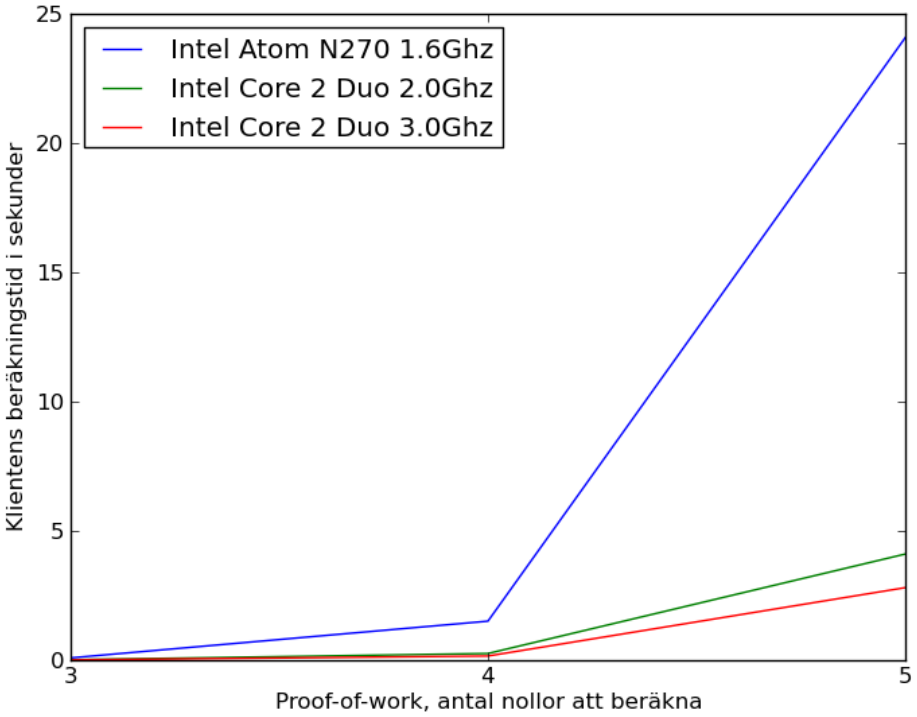
4.1.1 Pusslens svårighetsgrad

Som vi nämnde i del 2.2 så är det svårt att anpassa svårighetsgraden på pusslet i ett proof-of-work-protokoll. Pusslen måste vara svåra nog för att förhindra illasinnade användare från att överbelasta den skyddade nätverksresursen samtidigt som de inte får vara så svåra att en vanlig användare inte kan använda resursen. Eftersom en klient prövar sig fram till en lösning genom att beräkna ett hashvärde om och om igen påverkar antalet hashningar direkt hur lång tid som krävs för att lösa pusslet. Eftersom hårdvaran i datorer skiljer sig väldigt mycket mellan olika klienter så kommer olika klienter att få vänta olika länge på samma nivå av proof-of-work.

Ur figur 4.1 framgår det att processorhastigheten i stor grad påverkar den genomsnittliga tiden som krävs för en klient att lösa pusslet. Eftersom pusslet blir 16 gånger svårare för varje inledande nolla som servern kräver blir situationen snabbt ohållbar för klienter med långsamma processorer. Ett sätt att minska stegen mellan pusselnivåerna är att man godkänner lösningar som börjar på exempelvis tal mellan 0-7 istället för bara lösningar som börjar på 0, vilka blir enklare att hitta.

En möjlig åtgärd för att minska problemet med olika snabba processorer är att konstruera ett pussel som belastar minnet istället för processorn [4]. Internminnets hastighet har generellt mindre spridning än processorernas, men även minne kan ha tillräckligt stora skillnader i läs och skrivtider för att det ska bli lättare för vissa klienter än för andra. En annan lösning är att man skapar ett proof-of-work nätverk där man skickar delpussel till andra klienter som dessa måste lösa. Detta skulle innebära att en snabb dator fick mindre fördel gentemot andra [1].

Ett annat stort problem med pusslen är att variationen är stor mellan de uppmätta tiderna på samma svårighetsnivå, även när endast mätresultat från samma dator används. Det uppmätta fördelningen av hur många hashningar som krävs för

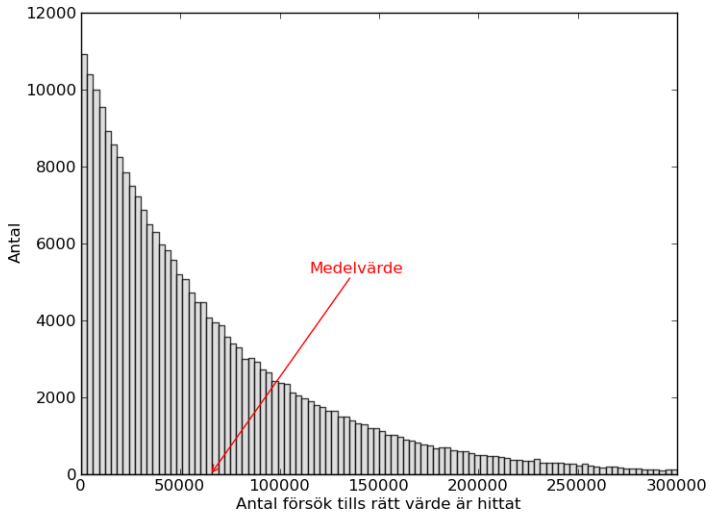


Figur 4.1. Tid att beräkna pussel för olika datorer

att lösa ett pussel presenteras i figur 4.2. Som synes finns det många klienter som får lättlösta pussel (tider nära noll), medans vissa klienter får väldigt svåra pussel (höga tider). En lösning för att komma runt detta problem är delpussel.

4.1.2 Delpussel

Eftersom klienten provar olika lösningar till dess att en korrekt lösning hittas kan man se antalet försök som en för-första-gången-fördelad stokastisk variabel med sannolikhet $1/16^d$, där d är antal inledande nollor i en godkänd lösning. För svårighetsgrad 3 blir sannolikheten för att hitta en lösning 1/4096 per försök. Det ger ett väntevärde på 4096 och en standardavvikelse som är lika stor. Detta är ett stort problem då en väldigt stor del av klienterna kommer lösa sitt pussel på en kort tid, en betydande bit under väntevärdet, medan ett fåtal kommer tvingas prova väldigt många olika lösningar, se figur 4.2. Ett proof-of-work pussel bör sträva efter att vara ungefär lika svår för alla klienter och därför måste standardavvikelsen vara betydligt mindre än väntevärdet.



Figur 4.2. Fördelning av antal försök på proof-of-work nivå fyra utan delpussel.

Pascalfördelning
$NB(r, p)$ där r är antal pussel och p är sannolikheten för att hitta en lösning.
Väntevärdet för fördelningen ges av: r/p.
Standardavvikelsen ges av: $\sqrt{\frac{r(1-p)}{p^2}}$.

För att uppnå en mer normalfördelad spridning på klienternas arbetsbörda vid beräkning av en lösning till ett pussel kan Pascalfördelningens egenskaper utnyttjas. Pascalfördelningen beskriver fördelningen av antal försök innan ett bestämt antal korrekta svar hittats, till skillnad från för-första-gången-fördelningen som endast beskriver fall där beräkningen stannar efter ett rätt svar [11, sid 434]. Detta betyder alltså att vi kan få en pascalfördelning om vi ger klienterna flera delpussel. Enligt pascalfördelningen egenskaper så kommer vi komma närmare och närmare en normalfördelning ju fler delpussel vi ger klienterna [11], se figur 4.3.

Vi har implementerat detta genom att tvinga klienten att beräkna ett flertal unika lösningar till samma pussel. Varje eftersökt lösning kan då ses som ett delpussel med en för-första-gången-fördelning. Servern belastas då endast med en extra hashning per delpussel som behöver verifieras.

Ur de framräknade (tabell 4.1) och uppmätta (tabell 4.1.2) värdena för antal försök som krävs innan klienten hittar en godkänd lösning till sitt proof-of-work-pussel framgår det tydligt att ett flertal delpussel av enklare svårighetsgrad är att föredra

Tabell 4.1. Beräknat antal hashningar för klienten.

r	d	**Väntevärde**	**Standardavvikelse**
1	3	4096	4096
4	3	16384	8191
8	3	32768	11584
16	3	65536	16382
1	4	65536	65536

Tabell 4.2. Uppmätt antal hashningar för klienten. Antal testfall varierar på grund av att körtiderna varierat och redovisas för att påvisa att resultaten är statistiskt säkerställda.

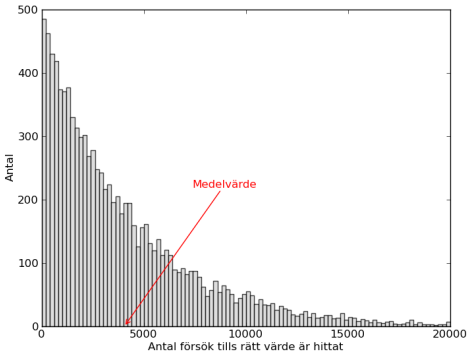
r	d	**Testfall**	**Medel**	**Median**	**Min**	**Max**	**Lägsta 10%**	**Högsta 10%**
1	3	10016	4067	2844	1	42376	428	9384
4	3	14617	16380	15157	958	73319	7130	27371
8	3	14049	32810	31605	5272	95241	19054	48158
16	3	83688	65539	64205	15473	160676	45596	87345
1	4	242597	65379	45328	1	849812	6863	150508

framför att använda sig av ett svårare pussel utan delpussel. Tabellerna visar även att gränsen till de lägsta respektive högsta tio procent av talen avviker mindre från medelvärdet i de fall där delpussel används. En tydligare bild av hur fördelningen förändras ges av figur 4.3 där man tydligt ser hur fördelningen centreras närmare medelvärdet.

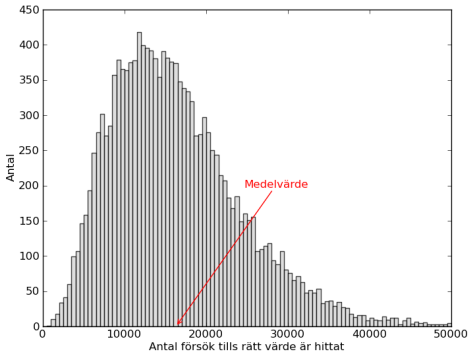
Flera enklare pussel ger som påvisat en jämnare fördelning av lösningstiderna med mindre chans för extremvärden. Ett problem med att använda sig av delpussel av denna typ är att en flerkärnig processor får ett uppenbart övertag gentemot en enkärnig processor i de fall klienten kan beräkna lösningar i flera trådar. Denna fördel förekommer dock även då endast ett svårt pussel utan delpussel ska beräknas eftersom klienten kan testa ett värde per processor på samma tid som en enkärnig processor testar ett värde och på så vis hitta en lösning snabbare.

4.1.3 Serverbelastning

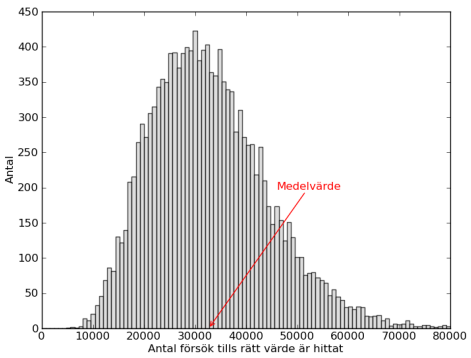
Under testerna har vi analyserat serverns prestanda genom att helt enkelt mäta hur stor processoraktivitet den har. Vi testade genom att attackera servern med fem datorer som alla ville ha tillgång till servern (de var alltså tvungna att lösa de pussel som de fick). När vi hade proof-of-work avaktiverat låg serverns processoraktivitet på 100% och servern slutade svara på vissa anrop. Klienternas processoraktivitet översteg inte 10% på någon av datorerna. När vi aktiverade proof-of-work så anpassade servern dynamiskt svårigheten vilket gjorde att den snabbt använde 1% av



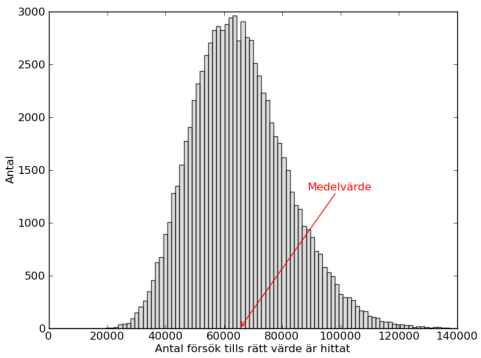
(a) Inga delpussel



(b) Fyra delpussel



(c) Åtta delpussel



(d) Sexton delpussel

Figur 4.3. Fördelning av beräkningsbörda vid olika antal delpussel på proof-of-work nivå tre.

processorkapaciteten medan alla klienter, oavsett hur bra hårdvara de hade, använde 100% av sin processorkapacitet. Detta visar vilken effekt proof-of-work kan ha för att reducera belastningen på en server.

Kapitel 5

Slutsatser

Proof-of-work kan skydda en bakomliggande resurs mot småskaliga attacker där attackeraren inte har tillgång till botnät eller kraftfull hårdvara, vilket visades när processorbelastningen flyttades från servern till klienterna under vårt belastningstest. Om en attackerare har tillgång till mycket processorkraft så kan man däremot ändå lösa så många pussel att proof-of-work svårighetsgraden höjs till en nivå där legitima klienter inte längre får tillgång till resursen utan att vänta orimligt lång tid.

Det är även möjligt att överbelasta servern genom att skicka en stor mängd förfrågningar till servern under en kort tid och på så sätt förhindra servern från att hantera legitima klientanrop. Detta problem är anledningen till att vi valt att använda proof-of-work på en server skild från den skyddade tjänsten. Genom att använda protokollet i de fall där det finns flera vägar in till en bakomliggande tjänst som samtliga implementerar proof-of-work kan den bakomliggande tjänstens tillgänglighet garanteras. Exempelvis i det fallet vi tidigare diskuterat där en mängd webbservrar som samtliga implementerar proof-of-work delar på en gemensam databas kan en eller flera webbservrar överbelastas på detta sätt. De attackerade webbservrarna kommer dock fortfarande inte släppa igenom för stor trafik till databasen och kommer därför inte riskera åtkomsten till den från de andra webbservrarna.

Hashberäkningsalgoritmen uppfyller de grundläggande kraven som ställs på en proof-of-work algoritm, se del 2.2. I vår analys visar det sig däremot att algoritmen i sitt grundutförande har en stor brist, nämligen att svårigheten för att lösa ett pussel kan variera stort mellan olika pussel på samma svårighetsgrad. För att komma runt detta problem så kan man använda delpussel, man sprider då ut sannolikheten att få ett svårt eller lätt pussel till ett mycket mindre spann som går mot att vara normalfördelat när antalet delpussel ökar. En annan nackdel med hashberäkning är att den är en mycket processorintensiv algoritm. Detta gör att en klient med snabb processor kan lösa ett pussel många gånger snabbare än en långsamare klient. En risk är att servern ökar upp proof-of-work-nivån under hög belastning till en nivå som inte påverkar klienter med bra datorer, men som ändå gör tjänsten oanvändbar för

klienter med gammal hårdvara. Detta skulle kunna göra att ekonomiskt missgynnade klienter skulle bli utelåsta, medans klienter med bättre förutsättningar skulle kunna använda tjänsten som vanligt.

Litteraturförteckning

[1] Marek Klonowski och Tomasz Struminski (2008). Proof of Communication and Its Application for Fighting Spam. *Proceedings of the 34th conference on Current trends in theory and practice of computer science, ISBN: 3-540-77565-X. Utgiven av Springer-Verlag Berlin*

[2] Cynthia Dwork och Moni Naor (1992). Pricing via Processing or Combatting Junk Mail. *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology, ISBN: 3-540-57340-2. Utgiven av Springer-Verlag London*

[3] Adam Back (2002). Hashcash - A Denial of Service Counter-Measure. *Rapport*

[4] Cynthia Dwork, Moni Naor och Hoeteck Wee (2005). Pebbling and Proofs of Work. *International Crytology Conference - CRYPTO, s. 37-54*

[5] Ben Laurie och Richard Clayton (2004). "Proof-of-Work" Proves Not to Work. *Presenterad på Third Annual Workshop on Economics and Information Security (WEIS04)*

[6] Willam Stallings och Lawrie Brown (2008). Computer Security - Principles and Practice, första upplagan, ISBN: 9780136004240. Utgiven av Pearson Education Inc.

[7] Thomas H. Cormen (2001). Introduction to algorithms, andra upplagan, ISBN: 0262032937. Utgiven av McGraw Hill.

[8] Johan Håstad (2000). *Notes for the course advanced algorithms.*

[9] Tuomas Aura, Pekka Nikander och Jussipekka Leiwo (2000). DOS-resistant authentication with client puzzles. *Revised Papers from the 8th International Workshop on Security Protocols, ISBN: 3-540-42566-7. Utgiven av Springer-Verlag London*

[10] Ari Jules, John Brainard (1999). Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks. *Publicerad i Proceedings of NDSS '99 (Networks and Distributed Security Systems), s. 151-165*

[11] Lennart Råde och Bertil Westergren (2003). Mathematics Handbook, femte upplagan, ISBN: 9789144031095. Utgiven av Studentlitteratur.