DoS Mitigation using Proof of Work

An example utilizing the UDP protocol

A N T O N L U N D Q V I S T
a n d J O N A T A N L A N D S B E R G

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2012

DoS Mitigation using Proof of Work

An example utilizing the UDP protocol

A N T O N L U N D Q V I S T
a n d J O N A T A N L A N D S B E R G

DD143X, Bachelor's Thesis in Computer Science (15 ECTS credits)
Degree Progr. in Computer Science and Engineering 300 credits
Royal Institute of Technology year 2012
Supervisor at CSC was Mikael Goldmann
Examiner was Mårten Björkman

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2012/landsberg_jonatan_OCH_lundqvist_anton_K12045.pdf

Kungliga tekniska högskolan
Skolan för datavetenskap och kommunikation

KTH CSC
100 44 Stockholm

URL: www.kth.se/csc

Abstract

In this report, we review the types of proof of work systems that already exist and are used in different environments. The main work that we have done is an implementation of our own protocol on top of UDP in order to make a server more secure against DoS attacks. After studying the results shown by the tests we found that there are good chances for a proof of work protocol to mitigate DoS attacks. We also show that it can not give complete protection as there will always be an upper limit on how much data a computer can handle.

Sammanfattning

I rapporten går vi igenom vilka typer av proof of work-system som redan idag finns och används i olika miljöer. Det stora arbetet vi själva utfört är en implementation av vårt eget protokoll ovanpå UDP i syfte till göra en server mer säker mot DoS-attacker. Efter att ha studerat de resultat som visats av testerna kom vi fram till att det finns goda chanser för ett proof of work protokoll att lindra DoS attacker mot servrar. Vi visar också att det inte går att få ett fullständigt skydd då det alltid kommer att finnas en övre gräns för hur mycket data som en dator kan hantera.

Statement of collaboration

The work has been divided equally between both the authors. Basically every row in both the report and the code has been either written, rewritten, commented on or discussed by both of the authors.

Contents

1 Introduction

Since the dawn of the Internet our dependency on services that use it for communication has escalated. We have reached a point where we become paralyzed if the services we use every day are not online. Essentials such as communication, payment, government services as well as all other applications we use can be overloaded and made unavailable with ease unless precautionary measures are taken. One way to remedy this is having clients that wish to connect to a server perform some work on their computer and show this to the server. The client will have to provide a *proof of work* (POW).

1.1 Background

Attacks aimed at overloading a server or service are called *denial of service* (DoS) attacks. When performed by multiple hosts simultaneously they are called distributed DoS (DDoS) attacks. Protecting yourself against DDoS attacks can be very difficult because of the sheer amount of data received. The attacks are also virtually impossible to protect your server from by just installing protective software or by configuring iptables because they will crumble under all the traffic. DDoS protection has to happen upstream by hardware that is load balanced and able to cope with up to tens of gigabits of data per second.[1] POW will not suffice as protection against DDoS.

DoS protection is perhaps more essential, because any client on the Internet can perform DoS attacks on their own. If you are not protected against DoS attacks and you use a protocol such as TCP (Transmission Control Protocol) you are vulnerable to attacks from any single computer. Having your service use an implementation of POW could aid with handling DoS attacks.

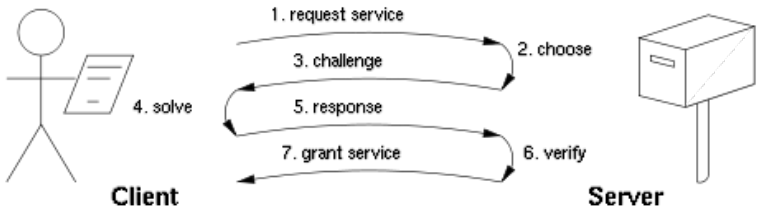
A core protocol driving communication on the Internet is TCP. It provides applications such as HTTP (the World Wide Web), BitTorrent, email etc. with reliable lossless transfer of data. When you want to send data to another host over TCP you have to perform a three-way handshake to establish a connection. First the initiating party, the client, sends a SYN-packet (SYNchronize) to the server telling the server it wants to connect. The server responds with a SYN-ACK (SYNchronize-ACKnowledgement) and finally the client responds with an ACK (ACKnowledgement) back to the server. After this a connection has been established and data can be sent.

However the three-way handshake is flawed. A common DoS attack is simply to flood a server with SYN-packets but never respond to the servers SYN-ACK-packets (TCP SYN flood). The server can have a limited number of outstanding SYN-ACKs and when the queue is full it will drop incoming SYNs. To anyone trying to access the server it will seem like it is unavailable. Because TCP was meant to be lossless and accommodate for e.g. slow clients, it will generally wait quite some time before dropping outstanding SYN-ACKs. Of course this time can be configured by server administrators but doing so could have ramifications on connectivity. Generating and sending enough SYN-packets to DoS a server is an easy chore for a computer. Implementing a POW into the three-way handshake could remedy exactly this. If a client has to perform some work before being allowed to connect it will not be able to overload the server as easily. The work per connection is very little but multiplied by a hundred thousand it will be a significant amount. This means that regular clients will not be affected by the POW, only clients who try to spam the server.

POW variants

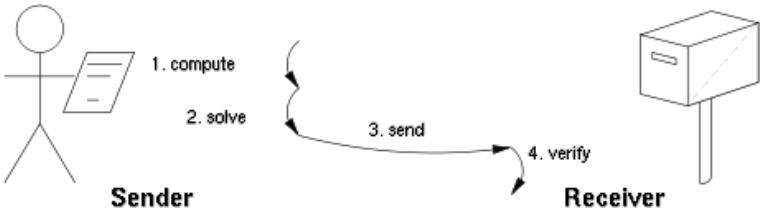
There are different ways to implement POW and different algorithms/puzzles to use for the actual work. The general idea of POW is that a client should perform work that can not be precomputed or faked. There are two ways to implement a POW protocol.

Figure 0.1. Challenge response process[7]



Challenge response. In a challenge response implementation the client wishing to connect to a server first asks the server for a puzzle or problem to solve. The client then solves the puzzle and sends it back to the client for verification. These events can be seen in Figure 0.1.

Figure 0.2. Solution verification process[8]



Solution verification. A solution verification implementation does not require the client to contact the server for a puzzle to solve. The work the client is supposed to do is predefined and the server only has to verify that the solution is valid for the data the the host for example wants to send. This is the case in email spam prevention algorithms such as Hashcash[6]. These events can be seen in Figure 0.2. POW to mitigate email spam does not prevent precalculation of solutions, but to send one million spam emails the sender would have to precalculate one million solutions.

Puzzle

Regardless of how you implement you POW protocol you need a puzzle algorithm. Puzzle algorithms can exhaust different resources of client computers. If an algorithm is CPU-bound it requires some CPU-time from the client. CPU-bound functions can vary greatly in execution time between computers with different performance.[5]. Hashcash is such an algorithm. In Hashcash the client has to concatenate a random number with a string several times and hash this new string. It then has to do so over and over until a hash beginning with a certain amount of zeros is found. Algorithms can also be memory bound. These algorithms strive to cause significant amounts of cache misses instead of just needing CPU time. Because the size of caches does not vary as much between different computers as CPU, this may be a better approach if you want puzzles that are solved equally fast on different computers.

1.2 Problem statement

With this report we wanted to answer the following questions:

- What are the principles of POW?
- Can our implementation mitigate a DoS attack?
- Does our protocol slow down communication significantly compared to a normal TCP connection?

2 Method

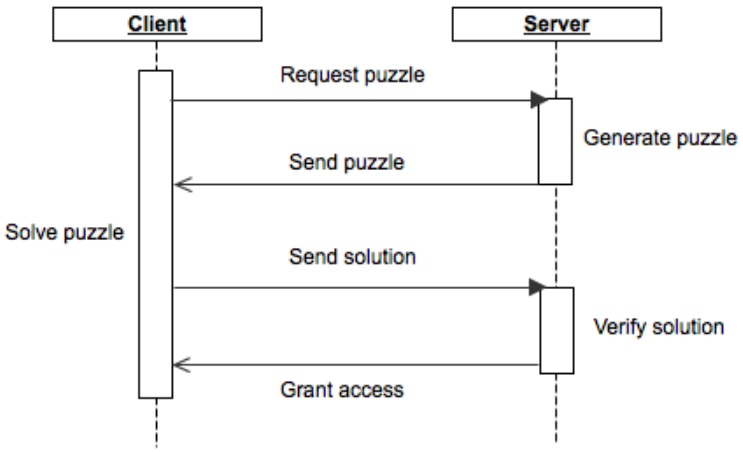
To answer the questions defined in the problem statement we needed we first read up on what has previously been done. We found no protocol that tried to mitigate DoS attacks or were adapted to work in the transport layer. This forced us to come up with our own version of a POW that we could implement over UDP[10] and that had the following characteristics:

- The workload on the server should be drastically lower than on the client
- The protocol needs to scale well when more clients tries to connect
- It should be impossible for a client to do any precalculations

2.1 The protocol

As shown in Figure 0.3, for a client to be allowed to open a connection over the TCP protocol it first has to solve a puzzle. A request for a puzzle is sent to the server over the UDP protocol. The puzzle is a SHA1[9] hash made up of the clients IP, the source port and a seed from the server. The server sends the puzzle to the client along with the corresponding seed id. The client then has to search for a string that when concatenated with the puzzle and hashed results in a string beginning with a fixed number of zeros. The solution is then sent back to the server along with the seed id.

Figure 0.3. Our version of a POW[4]



When the server gets a solution from the client it checks if the hash of the solution concatenated with the puzzle begins with the specified number of zeros. The puzzle is not stored anywhere on the server but since the client returns the seed id it can be reconstructed.

2.2 Implementation

Our implementation of a client and a server using our POW protocol can be found at `http://www.csc.kth.se/~antlu/pow.tar.bz2`

2.3 Testing

The different setups for testing were configured in a way so that one setup is with a strong server and a weak client. The other setups were done by increasing the performance of the client and decreasing the performance of the client.

Setup 1

- Client: Intel Core 2 Duo 1.3GHz. OS: BackTrack Linux 5.
- Server: Intel i5 2.3GHz hyperthreading. Only using one of the virtual cores with 50% execution cap. OS: BackTrack Linux 5.
- The client tries to spam the server with valid requests to the server, i.e. the client solves the puzzles.

Setup 2

- Client: Intel Core 2 Duo 1.3GHz. OS: BackTrack Linux 5.
- Server: Intel i5 DualCore 2.3GHz hyperthreading. Only using one of the virtual cores. OS: BackTrack Linux 5.
- The client tries to spam the server but does not care to solve anything. It only sends requests for new puzzles.

Setup 3

- Client: Intel Core 2 Duo 1.3GHz. OS: BackTrack Linux 5.
- Server: Intel i5 DualCore 2.3GHz hyperthreading. OS: BackTrack Linux 5.
- The client tries to spam the server but does not care to solve anything. It only sends requests for new puzzles.

Setup 4

- Client: Intel Atom 1.8GHz. OS: Ubuntu 11.10.
- Server: Intel i5 DualCore 2.3GHz hyperthreading. OS: BackTrack Linux 5.
- The client tries to spam the server but does not care to solve anything. It only sends requests for new puzzles.

The first step in setup 1 was to calibrate the number of required zeroes in the POW. This was done by measuring the time it took for the client to complete a request. When the time was lower than a tenth of a second we tried to overload the server with valid connections.

For this setup the source code was left nearly identical to how a regular client would perform a request. We programmed the client to send packets from 20 different ports. For each port it sent 15 packets, i.e. requested 15 identical puzzles and solved them. Thus resulting in 300 requests.

With setup 2, 3 and 4 we tried two different scenarios. The first was when the server handled all the requests but returned empty puzzles. The second scenario was when the server calculated new puzzles and returned them to the client.

The source code of our working implementation was edited to allow the client to spam the server, the client did not have to calculate any hashes, i.e. perform POW. We configured the client to send 1000 packets per port from 380 ports, meaning that the client would attempt to send 380,000 puzzle requests.

Tcpdump was used in all setups to count packets and we ran it with the following flags: `tcpdump -n -q -t -s 100 -w output`. Two instances of tcpdump ran simultaneously on both the server and the client. The first one had a filter looking for the host destination ip and the other had a filter looking for the host source ip. This was done just before running `client` and `server`. Between runs the tcpdump processes were terminated and the number of packets captured by the filters noted. The time was measured by running the client with `time client`. The real time was then noted. CPU peaks were checked using the program `top`.

3 Results

In this section we present the data obtained from our tests. We chose to not write any further description or analasys of the data here. To get a good grip of what this data says the reader is encouraged to continue to read in Section 4.1 where we make references to the figures.

Table 0.1. Result from setup 1

run nr	time	client CPU peak	server CPU peak	pkts sent by client	pkts rec by server	pkts sent by server	pkts rec by client	pkts handled in server
1	14,517	100	30	600	600	300	300	600
2	4,776	100	60	600	600	300	300	600
3	11,807	100	47	600	600	300	300	600
4	21,832	100	33	600	600	300	300	600
5	10,019	100	64	600	600	300	300	600
6	12,829	100	41	600	600	300	300	600
7	9,198	100	43	600	600	300	300	600
8	10,862	100	47	600	600	300	300	600
9	14,569	100	38	600	600	300	300	600
10	9,784	100	75	600	600	300	300	600
Average	12,019	100	48	600	600	300	300	600

Table 0.2. Result from setup 2

Scenario 1 - empty puzzles

run nr	time	server CPU peak	pkts sent by client	pkts rec by server	pkts sent by server	pkts rec by client	pkts handled in server
1	20,516	85	60 114	60 114	15 491	15 491	35 205
2	20,462	82	67 556	67 556	15 494	15 494	32 071
3	20,494	81	70 129	70 129	15 493	15 493	34 655
4	20,441	87	35 766	35 766	15 490	14 813	24 379
5	20,511	85	52 898	52 898	15 490	14 927	40 614
6	20,457	90	50 150	50 149	15 490	15 490	34 776
7	20,391	91	44 980	44 979	15 490	15 271	30 782
8	20,450	82	53 289	53 289	15 490	15 490	15 490
9	20,474	89	51 607	51 605	15 490	15 490	31 910
10	20,429	86	45 216	45 216	15 490	15 253	33 889
Average	20,463	86	53 171	53 170	15 491	15 321	31 377

Scenario 2 - real puzzles

run nr	time	server CPU peak	pkts sent by client	pkts rec by server	pkts sent by server	pkts rec by client	pkts handled in server
1	20,480	95	33 955	33 948	15 492	14 606	16 911
2	20,461	82	61 914	61 906	13 135	13 135	13 135
3	20,475	84	54 438	54 432	13 886	13 886	13 886
4	20,479	92	32 183	32 177	15 492	14 700	15 534
5	20,445	85	52 280	52 274	13 476	13 476	13 476
6	20,484	86	54 807	54 800	13 879	13 879	13 879
7	20,486	93	35 714	35 714	15 492	15 045	17 130
8	20,511	93	41 128	41 120	15 145	14 902	15 145
9	20,430	92	35 986	35 979	15 327	14 312	15 327
10	20,460	87	49 346	49 340	14 188	14 188	14 188
Average	20,471	89	45 175	45 169	14 551	29 777	14 861

Difference between scenario 1 and 2

	time	server CPU peak	pkts sent by client	pkts rec by server	pkts sent by server	pkts rec by client	pkts handled in server
	0,042	3	41	47	939	14 524	19 028

Table 0.3. Result from setup 3

Scenario 1 - empty puzzles

run nr	time	server CPU peak	pkts sent by client	pkts rec by server	pkts sent by server	pkts rec by client	pkts handled in server
1	20,564	38	55 155	55 155	32 284	31 068	55 155
2	20,580	42	56 589	56 589	32 284	24 037	56 589
3	20,514	49	54 939	54 939	32 284	26 428	54 939
4	20,548	50	56 812	56 812	32 284	30 005	56 812
5	20,573	48	58 126	57 534	32 284	25 147	57 534
6	20,556	39	51 394	51 394	32 284	32 275	51 394
7	20,579	41	47 286	47 286	32 284	28 412	47 286
8	20,523	42	49 022	49 022	32 284	31 080	49 022
9	20,568	42	49 677	49 666	32 284	30 607	49 666
10	20,529	43	52 258	52 051	32 284	25 682	52 056
Average	20,553	43	53 126	53 045	32 284	28 474	53 045

Scenario 2 - real puzzles

run nr	time	server CPU peak	pkts sent by client	pkts rec by server	pkts sent by server	pkts rec by client	pkts handled in server
1	20,550	51	35 295	35 294	32 308	30 895	35 294
2	20,554	56	38 813	38 813	32 308	30 542	38 813
3	20,492	61	43 535	43 535	32 308	25 824	43 535
4	20,540	59	46 173	46 171	32 286	30 552	46 171
5	20,492	66	53 432	53 432	32 286	23 845	53 432
6	20,568	79	56 313	53 612	32 286	29 333	53 467
7	20,601	64	48 100	48 099	32 286	28 785	48 099
8	20,560	65	52 100	52 099	32 286	31 798	52 099
9	20,549	63	49 066	49 066	32 286	31 852	49 066
10	20,553	71	53 930	53 930	32 286	25 204	53 930
Average	20,546	64	47 676	47 405	32 293	28 863	47 391

Difference between scenario 1 and 2

	time	server CPU peak	pkts sent by client	pkts rec by server	pkts sent by server	pkts rec by client	pkts handled in server
	0,008	20	5 450	5 640	9	389	5 655

Table 0.4. Result from setup 4

Scenario 1 - empty puzzles

run nr	time	server CPU peak	pkts sent by client	pkts rec by server	pkts sent by server	pkts rec by client	pkts handled in server
1	20,203	78	379 680	379 675	32 283	32 284	76 831
2	20,206	79	379 645	379 639	32 283	32 283	106 766
3	20,198	71	379 585	379 585	32 283	32 271	90 100
4	20,198	77	379 631	379 611	32 283	32 283	116 103
5	20,192	74	379 654	379 652	32 283	32 281	120 834
6	20,244	76	379 838	379 826	32 283	32 282	116 668
7	20,188	72	379 654	379 642	32 283	32 283	100 004
8	20,187	74	379 609	379 609	32 283	32 283	103 135
9	20,200	77	379 681	379 676	32 283	32 282	100 450
10	20,189	78	379 568	379 559	32 283	32 282	96 598
Average	20,201	76	379 655	379 647	32 283	32 281	102 749

Scenario 2 - real puzzles

run nr	time	server CPU peak	pkts sent by client	pkts rec by server	pkts sent by server	pkts rec by client	pkts handled in server
1	20,187	79	379 649	379 645	32 283	32 283	74 046
2	20,193	78	379 604	379 604	32 283	32 283	65 883
3	20,194	79	379 606	379 605	32 283	32 283	76 195
4	20,194	77	379 681	379 680	32 283	32 283	73 557
5	20,193	79	379 629	379 629	32 283	32 283	80 445
6	20,192	77	379 655	348 074	32 283	32 283	59 844
7	20,192	78	379 660	379 660	32 283	32 283	79 107
8	20,206	79	379 805	379 790	32 283	32 283	78 233
9	20,189	79	379 607	379 607	32 283	32 283	76 204
10	20,198	79	379 696	379 695	32 283	32 283	76 011
Average	20,194	78	379 659	376 499	32 283	32 283	73 953

Difference between scenario 1 and 2

	time	server CPU peak	pkts sent by client	pkts rec by server	pkts sent by server	pkts rec by client	pkts handled in server
	0,007	3	5	3 149	0	2	28 796

4 Discussion

4.1 Analysis of results

The purpose of the first setup was to verify that the protocol worked as expected when a host tried to spam the server with legitimate traffic. In setup 1 we used a quite slow server and a moderately good client. As can be seen in table 0.1 the client CPU peak average was 100% while the server peak average was 48%. The client has to work very hard to send 300 legitimate packets to the server. The reason that the times are quite scattered is because a random number is generated and then incremented until a solution is found. The average time to solve all puzzles is approximately 12 seconds. During these 12 seconds the client sent 300 requests, solved 300 puzzles and sent 300 solutions to the server. The client sent $300/12 = 25$ packets per second, alternatively it took $12/300 = 0.04$ seconds to send a single packet. The measured times are true for the client computer used in setup 1. If the client is run on a better computer it will produce packets faster. This is a drawback of using a CPU-bound POW as we have done. Alternatives are presented in Section 1.

When choosing a way to measure the CPU load during the tests the best way would be to calculate the area under the graph CPU load versus time. An easier measurement and the one that was available to us was to read the peak of the graph. In setup 2, 3 and 4 this described very well the overall load on the server CPU but not in setup 1. During the major part of each test with that setup the server CPU load increased by only a few units from normal use with the exception of one short peak.

In tables 0.2, 0.3 and 0.4 there is a lot of packet loss at different places. The packets that are received by the server but not taken care of are probably due to a poor implementation of the server. It is possible that a better implementation could fetch data faster from the network card's buffer. Although it would let the server handle more packets there will always be a limit to the number of packets it can cope with. Some packets are also lost when the server tries to send puzzles back to the client. This could also be the result of us overflowing the network card's buffer. We believe that the small differences in packets sent and received (differences of less than 100) are due to the unreliable nature of UDP. At every hop the packets run the risk of being dropped and since these setups were meant for spamming we commented out our simple resending function.

The time in setups 2 and 3 is constant. This is because we try to send a fixed number of packets, 380,000, every time we ran the client and because it ran on the same computer. In setup 4 the same client ran on a different computer and it went marginally faster. The client computer in setups 2 and 3 has a faster CPU than the computer in setup 4 and hence should execute faster. The reason for this is unknown.

The most interesting data is in the differences between the two scenarios in setups 2, 3 and 4. In these tables we are able to see how much POW affects the performance of the server. The server CPU in setup 2 and 4 is nearly the same with and without POW. It was unable to handle all packets even before POW was turned on but the amount of packets not handled increased when POW was turned on. This is an indicator that the server was already working at it is max capacity before POW was turned on. It also confirms that POW affects the performance negatively as the server handles even less packets with POW.

A close look at setup 3 reveals that no packets were dropped by the server, i.e. all packets received were handled. In this setup we can also see a clear CPU increase when we turned on POW. This is very likely because the server in setup 3 was powerful enough compared to the client that the server had no problem coping with the load produced by the client. There was even enough CPU left to accommodate the POW hashing.

4.2 Conclusions

Hellman and Rios [2] wrote in the beginning of their report that POW should not be used in systems that rely on fast request processing since the main principle of POW is that extra work needs to be done. As one can see in the analysis of the results the time it takes for a client to do POW when opening a web page can be up to 0.24 seconds. For most people this wont be noticeable. If a system requires more focus on speed than on security it is easy to lower the number of required initial zeros. For more suggestions see Section 4.3.

By using the POW protocol the limit on how many incoming connections the server can handle is transfered to the CPU. In a normal setup the limit is memory bound and limited to the number of available slots. It is therefore possible that POW could mitigate a DoS attack.

Setup 1 also shows us that it is difficult for a client to overload a server when it provides a POW. This gives us some proof that a POW can work as a protection for the TCP slots. A possible setup is a firewall that only allows connections to pass through if the client provides a POW.

The conclusion one can draw from setups 2, 3 and 4 is that POW is not bulletproof. No matter how you build your system it will be vulnerable to too much traffic. All packets coming in to a device must be handled in some way so it is just a matter of sending a large enough amount of packets.

4.3 Further research

Here we present a few ideas on what could be interesting to do if one wants to continue upon the research that we have begun.

In the current implementation of the protocol the puzzle includes a static seed. The idea from the beginning was that the server should continuously create new seeds to make any precalculations impossible for a client. For this to work the server has to keep a short history of the most recent seeds so that even though new connections are using new seeds a client still has the possibility return a solution with an old seed. By defining the length of the history one also defines the time a client has to solve a puzzle before it is invalidated.

In this report we have chosen to not discuss how the server and client should continue their communication after the POW is provided. There are many possible ways one can think of and each have their pros and cons. The scenario we have talked about earlier is to implement a POW system into a firewall. This is an important step when evaluating if POW is effective as a means of DoS mitigation.

As written earlier in Section 4.1, using 16 as the required number of initial zeros is not always satisfactory. The situation could require either faster communication or stronger protection against attacks. Another example of how to comply with this varying need is suggested by Kaiser and Feng[3]. The idea is that you vary the difficulty of the puzzle depending on server load and how fast a client has solved earlier puzzles. This could easily be done by not letting the number of initial zeros be a fixed value but rather to tell the client on each request how many zeros it should find.

Bibliography

[1] Darren Anstee. DDoS Attack Trends Through 2010, Infrastructure Security Report & ATLAS Initiative. Available from `http://ripe62.ripe.net/presentations/88-Darren-Anstee-AA-RIPE-2011-DDoS_Trends.ppt.pdf`, 2010. [cited: 2012 april 12].

[2] Christian Hellman and Felix Leopold Rios. Proof of Work. Available from `http://www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2010/rapport/hellman_christian_OCH_rios_felix_leopoldo_K10039.pdf`, 2010. [cited: 2012 March 2].

[3] Ed Kaiser and Wu chang Feng. mod kaPoW: Mitigating DoS with Transparent Proof-of-Work. Available from `http://conferences.sigcomm.org/co-next/2007/papers/studentabstracts/paper46.pdf`, 2007. [cited: 2012 March 6].

[4] Anton Lundqvist and Jonatan Landsberg. File:Proof of Work protocol for DoS Mitigation. Available from `http://www.nada.kth.se/~jonlan/kex/clientserver.png`, 2012. [cited: 2012 may 20].

[5] Aron Sharma and Damon Sharivar. Proof-of-Work. Available from `http://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand11/Group8Mikael/final/Aron.Sharma.Damon.Shahrivar.report.pdf`, 2011. [cited: 2012 March 5].

[6] Wikipedia. Hashcash [homepage on the Internet]. Available from `http://en.wikipedia.org/wiki/Hashcash`, 2011. [cited: 2012 april 12].

[7] Wikipedia. File:Proof of Work challenge response.svg. Available from `http://en.wikipedia.org/wiki/File:Proof_of_Work_challenge_response.svg`, 2012. [cited: 2012 april 12].

[8] Wikipedia. File:Proof of Work solution verification.svg. Available from `http://en.wikipedia.org/wiki/File:Proof_of_Work_solution_verification.svg`, 2012. [cited: 2012 april 12].

[9] Wikipedia. SHA-1 [homepage on the Internet]. Available from `http://en.wikipedia.org/wiki/SHA-1`, 2012. [cited: 2012 may 20].

[10] Wikipedia. User Datagram Protocol [homepage on the Internet]. Available from `http://en.wikipedia.org/wiki/User_Datagram_Protocol`, 2012. [cited: 2012 april 12].