DISPUTE: Distributed Puzzle Tussle

A. Lukyanenko†, A. Gurtov‡, A. Ylä-Jääski†
† Department of Computer-Science and Engineering, Aalto University, Finland
‡ Helsinki Institute for Information Technology HIIT, Aalto University, Finland

***Abstract*—Distributed Denial of Service (DDoS) attack continues to be one of the main vulnerabilities of today's Internet. Client's puzzle mechanism is a well-known solution against such threat, however with badly tuned puzzle sizes it may harm the clients in the peaceful time, as well as produce additional difficulties during an attack.**

Here, we introduce a novel algorithm — DISPUTE — auto-tunable distributed puzzle mechanism with variable puzzle sizes. Main feature of it is that the server does not need to adjust any puzzle sizes, instead the clients during the "fight for" server resources find some form of equilibrium situation on the server side.

We describe the algorithm and show the DISPUTE's performance using a simulation tool. The results suggest that regular (laptop) users, as well as light (sensor) users can successfully access a server even during a heavy DDoS attack.

I. Introduction

Recent decade showed the Internet's vulnerabilities towards distributed denial-of-service (DDoS) attacks [1]. This is especially true for some protocols, which have design weaknesses, such as memory limitation (TCP SYN) or CPU limitations for enormous request coming during a short period of time. The threat is strengthened by the fact that a lot of machines connected to the Internet are infected with Trojans and viruses, and therefore unconsciously become a part of the mass DDoS attacks on different servers. In the News we see a lot of reveled events concerning huge botnets of sizes even 13M stations [2].

Recent decade also produced a set of solutions against DDoS attacks [1], including (but not limited to) overlay mechanism (e.g., i3 [3]), new Internet architectures (e.g., DONA [4]) and client's puzzle mechanisms [5], [6]. The latter is probably best applicable for today's Internet and does not require new Internet architecture

The client's puzzle mechanism is basically very simple. A client asks a server for some service. The server before starting processing the client, generates some random number X and based on it using hash-function (for example, MD5), produces hash operation on it Y=MD5(X). After that it defines on the difficulty of the puzzle (K) and produce modified X value: $X' = X >> K$, where $>>$ is right bit shift operation. The server replies to the client with (X', Y, K) packet. The client has to find original X and inform the server the solution. As it can be noticed server side operations are generally very cheap, while the client side operation is to reverse the hash-function, by the hash functions design it is not a chip operation and produced by brute-force search. Thus, in average, with assumption of no other collisions happen, it will take 2^{K-1} MD5 computations, which result in corresponding time.

The client's puzzle mechanism may be applied in transparent way and one of such transparent solutions is Host Identity Protocol(HIP) [7], [8], which has puzzle algorithm implemented inside. The HIP's puzzle mechanism produces one per connection solution in the base exchange (BEX) packet I2, it precedes to another BEX packet — R1, which request for the puzzle with given K.

The puzzle mechanism is mainly implemented with support of adjustable difficulty value (K), indeed it is not worth to ask huge puzzle sizes with almost empty server, and it is worth to increase K whenever the attack on the server is being strengthened. However, in practice, the puzzle sizes are often fixed to some predefined values. The connection between the volume of current server load and the puzzle size is also often lost. All the above result in inefficient usage of clients CPU and battery.

With such prefixed K values, optimized for the "normal users", i.e., home PCs, the puzzle mechanism produce harm for the limited-CPU (and/or battery) devices, for example, mobile phones or sensor devices. The work [9] moreover concludes that the use of puzzle on sensor is not feasible at all.

Finally, recent news has revealed a scenario when an attacker my start to produce a DDoS attack using new "cheap" technologies, such as cloud infrastructure [10].

In order to deal with more powerful attacks the server needs to increase K, as was said it may harm the hand-held devices if the increase is rapid using some prefixed values. The equilibrium for the puzzle sizes is needed.

In this work we introduce a novel protocol — DIStribute PUzzle TusslE (DISPUTE), which has the following natural properties:

1) No need for the server to explicitly set the K value.
2) Natural equilibrium between clients and the server is formed.
3) Clients can easily parallelize the load among the helping nodes.
4) The puzzle computation task requires less signaling traffic, and thus, less battery exhaustion.

II. Algorithm and Implementation

In this section we are going to explain the DISPUTE algorithm, with its properties.We will talk a little about the implementation issues and the features which it has.

A. Basic Algorithm

The DISPUTE algorithm is an extended version of MKFS algorithm [11], where we may assume that K always equals

to 0. The MKFS was producing the fight for the resources by the principle of Defense-through-Offense [12], sending a lot of "basic entrance" packets. We extend the idea with utilizing puzzle computation inside these "entrance" packets. Here the puzzle size K is variable. Server may decide on the minimal/maximal value available for K, however only client should decide what size of the puzzle it wants to solve. The client also is allowed to compute additional puzzles if it is not enough yet. The total amount of the resources the client spent to solve the puzzles are accumulated on the server side.

Having variable K value, we achieve the situation that different clients spent different CPU time for the solutions, and thus a priority is needed to give the resources to the clients which used CPU most in total up-to-the-moment. It is fair to serve the one who spent the most and this is the main idea of the DISPUTE algorithm. Such usage of the DISPUTE algorithm is novel to our knowledge, and latter we will explain what beneficial properties does it carry.

The DISPUTE algorithm consists of two parts — one on the Server's side, another one is on the Client's side. The server side is basically a work of a special structure main part of which is a simple heap-structure [13]. The server has to be able to produce two procedures: *add puzzle solution information to the structure*, and *pop the top element from the structure*. Both of them are based on priority queue structure (heap based) – $queue$, and a small database of the communicated clients – $id\_database$. Priority queue provides access to the maximum value with complexity $O(1)$, and update operation with complexity $O(\log n)$, while $id\_database$ provides fast access to the $queue$ elements by client's IDs with complexity $O(1)$. However, the $id\_database$ should update its pos field every time the $queue$ is being updated.

Algorithm 1 Adding puzzle computation to the DISPUTE queue algorithm.

Input: $queue$ – is a priority queue organized as a heap; $id\_database$ – is a map $id \to pos$; pos – position in the queue; id – is the user ID to add, and K– puzzle size computed.

```
if id ∈ id_database then
    pos ← id_database[id].pos
    queue[pos] ← queue[pos] + 2^K
    Update_heap_and_database(queue, id_database, pos)
else
    queue.push_back(id, 2^K)
    pos ← queue.size
    id_database ← {id, pos}
    Update_heap_and_database(queue, id_database, pos)
end if
```

The first algorithm is depicted as Algorithm 1. The server receives computed and verified puzzle of size K from the client id. Based on this event the server calls function $add(id, K)$. First of all, the function checks whether the id value is present in $id\_database$. If not, it creates new $queue$ elements and appends it to the $queue$, at the same time it adds information on the new client to a new $id\_database$ element. After that the function starts the procedures to put the element into correct place of the heap ($queue$), with correctly updated database ($id\_database$). If the id is the client which we already were communicated with, then using $id\_database$ element of the id we find its position in the queue and updating the $queue$ element by the computation efforts produced by client id.

Algorithm 2 Taking maximum from the DISPUTE queue algorithm.

Input: $queue$ – is a priority queue organized as a heap;$id\_database$ – is a map $id \to pos$; pos – position in the queue;

Output: id – client ID to be served

```
if queue is not empty then
    pos ← 1
    id ← queue[1].id
    queue[pos] ← queue[queue.size]

    id_database[id].clear()
    id_database[queue[1].id].pos ← 1
    queue.remove_last()
    Update_heap_and_database(queue, id_database, pos)
    return id
end if
```

The second algorithm is depicted as Algorithm 2. It is called every time the server is able to start serving a new client. The server simply calls function $pop()$ for the structure and should receive id of the element in the top of the $queue$ or -1 if there is no element, i.e., the $queue$ is empty. In case the $queue$ is not empty the function produces basic heap algorithm: takes the top of the queue, erases it, places the last element of the queue on the top place, reduces the queue, and updates it starting from the top element. We additionally in the function erasing the database element connected to the returning id.

Finally, notice that the algorithms uses the priority computed based on the puzzle sizes K. It is equal to 2^K. We have already mentioned in the Introduction how this value is connected to puzzle size. More on why we decided to use this weight function for given K will be explain in the Simulation section of the paper.

B. Client's Strategies

The server-side algorithm is fixed and does not allow deviations. The client part of the DISPUTE algorithm is undefined. A client may ask to solve any puzzle size from the start, however, probably the best is to ask for the minimal one. The probability the given server is under the DDoS attack is not so high generally. It is worth to send testing message which reveals whether the server is loaded or not. This part of the protocol may even be enhanced further (see the next Section). After that some increasing procedure for the requested K value may be used.

In the work we are not going to study all possible strategies or even any sophisticated once. During the Simulations, we are

going to use the idea that the client id simply decides what K_{id} to employ during the whole communication round. However, we will have different clients with different K values and thus we will get which is better to use under which conditions.

Furthermore, we want to note that the whole communication scheme resembles a first-price repeating auction or a multiple-object first-price auction [14]. We have non-classical incremental bid scheme, however, the theory on the first-price auctions says that the identical bidders will form same valuation of an object, which may be less than the "true-value" of the object. In our case, however we do not aim to "sell" the service as high as possible (i.e., we do not want all the clients to use as much CPU as possible). Thus, the equilibrium which is formed based on first-price auction is fully desirable for us. On the other hand the equilibrium is minimal bid the clients have to pay for the service. In such case the first bid equal to the minimal K is reasonable if the probability that the fixed server is under attack at a given moment of time is small enough.

The DISPUTE algorithm also has a legacy property from MKFS algorithm, even with minimal puzzle computation the client will at some time enter the system. The time required for the client to enter the system for a scheme without puzzles is studied in [11].

C. Alternative solutions

The DISPUTE algorithm that was suggested is not an only solution to the problem of dynamic server load control with the use of puzzles. There is always exist solutions without usage of any queue, i.e., send out a puzzle of size K to the client, where K is chosen by the server depending on the current observed load.

Let us take a closer look on these kinds of solutions. As the server does not maintain any queuethen it has an obligation to serve a client if it solves the previously given puzzle. Thus the based on the load, and the trend the server has to predict own load at the moment when it will receive the solution.

There are possible two cases:

1. The server simply gives the puzzles according to the current own load without certain prediction based on requests, then zombies may come in a "crowd" way ask for puzzles. The server will give them simple one, without prediction, and after solution zombie machines will send a lot of solutions at once. The latter will make the server (a) start serving a lot of zombie machines unexpectedly, and (b) make high puzzle solution for new-coming benign clients.

2. The server starts to predict the new future load based on number of puzzle requests. It means that it will give to every next client higher value for the puzzle to solve (if there is no leaving clients). In that case the zombie stations may ask for the new puzzles, but do not even try to solve it, the increasing strength of the puzzle will harm the benign clients.

Considering to the above mentioned cases the DISPUTE algorithm is free from these problems, as there is no need for prediction for the load, nor obligation to serve the clients based on the first puzzle solution.

III. Simulation

In this section we are going to discuss the implementation setup and what tests were used, after that we will present our experimental result.

A. Implementation

In order to make the testing work faster we implemented our own network simulator in C++ (1500 lines of code + scripts). It has all the standard logic of a discrete-event network simulator. The main entities for the simulator are client bundles and the server. The client bundles consist of some number of homogeneous stations attempting to help the main client to enter the server in an individual manner. All the stations receive their power value explicitly from the input file and requesting puzzles from the server, producing solutions in response. If the number of stations in the bundle is fixed to one, then it corresponds to one individual station attempt to get into the server.

The server, in turn, provides all necessary DISPUTE functionality, it has the data structure, and maintains the priority queue in it, answering only to the elements which are in the head of the queue.

In order to test DISPUTE we modelled a network, where the server connected to the bundles of clients by a network with 100ms link delay with 10% fluctuation. The bandwidth was not considered as a problem. Every station in every bundle follows a simple strategy: it chooses the size of the puzzle K at the start of the run, after that during the run of simulation it sends a request message for a puzzle of size K, solves using own capacity and sends the answer to the server. Every station from the same bundle says the id of main station from the same bundle. After receiving an answer from the server, stations of a bundle starts a new attempt to enter the server (this delay-synchronized strategy and in next section we will explain why we still use it).

Using the network simulator we created a number of topologies and tests. In result some of the tests were done for fixed topology in order to show how different stations behave and receive attempting to connect to the server in parallel. From now on we will use the following notations for the stations and strategies: I, L, S stand for Imote2 sensor, laptop PC and hand-held smart device (our interpolation with $p = 40$). The underline number goes for the chosen strategy, for example I_5 that Imote2 sensor requests puzzles of size 5, and L_{20} for the laptop requesting puzzle of size 20. The multiplier goes for the number of devices in a bundle. For example, $3xS_5$ is for three smart devices which use strategy 5.

B. Results

For all the tests we fix the zombie stations to be laptops-capable PCs with strategy 10. This suits perfectly to our test purposes, as the number of zombies variable and having 1000 zombies with strategy 10 is equivalent to having 500 zombies with strategy 20 (ignoring the latency issues).

For the first test, we decided to fix the zombie size $Z = 1000$ and make the variable the size of the bundle for

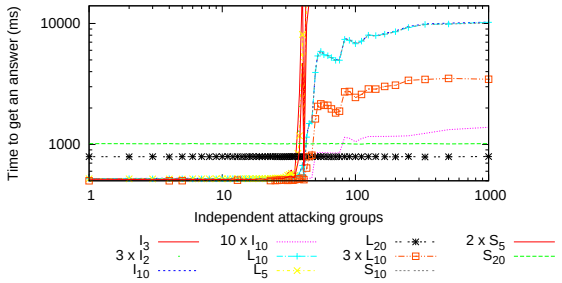

Fig. 1. Puzzle computation time for fixed topology with variable distribution of attackers onto subgroups.

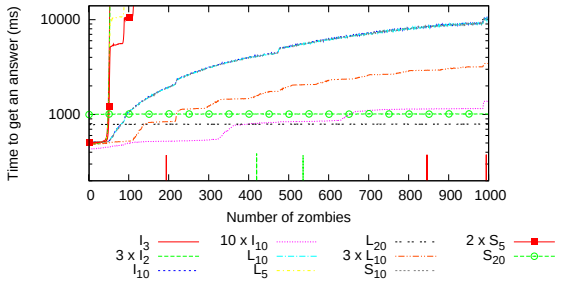

Fig. 2. Puzzle computation time for fixed topology with variable number of attacking zombies.

zombies, the zombie stations may be divided to two bundles of size 500 each, or may be four bundles of size 250. As we have simplistic strategy here for zombies, the division is very important in order to maximize the attack power on the server. The Fig. 1 depicts our simulation results for this situation, on the X axis is the number of groups or bundles on which the attacking stations are divided, on Y axis is the time needed for benign clients of fixed topology to get an answer from the server.

As we can see independently of the clients strategies the server produce higher attack if separates attacking nodes as much as possible. From now on, we will assume that the attacker will always use this kind of strategy.

Secondly, in Fig. 2 again for the fixed topology we produce an attack, but now we variate the actual number of zombies. Starting from 0 up until 1000 and again produce delay to get an answer. As we can see, at the start (when the number of zombies is small) for the benign clients it is good to chose strategy with small puzzle sizes K, however as the attack strengthen the decision to chose higher puzzle size becomes dominant. Additionally, it reduces the number of communication and, hence, battery usage, and beneficial for sensor devices. The devices S_{20} and L_{20} has almost constant time to get served all the time. This is because they chose the puzzle size which is higher than average top of DISPUTE queue and one attempt (and one solution) is enough to get a reply. The different between these devices is only because S_{20} is less powerful than L_{20}. Additionally, 10 I_{10} devices spend almost the same time to get a reply as S_{20} device. This shows how cooperation of small devices can help them get an answer during DDoS

IV. Conclusion

In this work we introduced a novel dynamic puzzle size algorithm — DISPUTE, which is auto-tunable due to the fact that clients play the role of a puzzle size selector rather than the server. This leads by usage of first-price auction scheme close to the "true price" computation efforts needed for every station. Finally, DISPUTE protocol is analyzed with the help of a custom constructed simulator.

For the future work on the DISPUTE protocol we leave the detailed analyses of CPU time influence by the K value, and construction on it basis a more correct simulator, with enhanced set of strategies for the end clients. Then the clients can find the optimal value for K instead of fixing one value for the whole period of time. Additionally, real-world experiments would help to justify this protocol benefits.

Acknowledgment

This work was supported by the Academy of Finland, grant number 135230.

References

[1] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 39–53, 2004.

[2] A. P. Jordan Robertson, "Authorities bust 3 in infection of 13m computers." http://www.usatoday.com/tech/news/computersecurity/2010-03-02-botnet-arrest_N.htm.

[3] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana, "Internet indirection infrastructure," in *SIGCOMM '02: Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 73–86, ACM, 2002.

[4] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica, "A data-oriented (and beyond) network architecture," in *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, (New York, NY, USA), pp. 181–192, ACM, 2007.

[5] A. Juels and J. Brainard, "Client puzzles: A cryptographic countermeasure against connection depletion attacks," in *NDSS '99: Proceedings of 1999 Network and Distributed Systems Security Symposium*, (San Diego, CA, USA), pp. 151–165, Internet Society, 1999.

[6] T. Aura, P. Nikander, and J. Leiwo, "DOS-Resistant Authentication with Client Puzzles," in *Security Protocols* (B. Christianson, J. Malcolm, B. Crispo, and M. Roe, eds.), vol. 2133 of *Lecture Notes in Computer Science*, pp. 170–177, Springer Berlin / Heidelberg, 2001.

[7] A. Gurtov, *Host Identity Protocol (HIP): Towards the Secure Mobile Internet*. Wiley Publishing, 2008.

[8] "Host Identity Protocol (HIP) IETF working group." http://www.ietf.org/dyn/wg/charter/hip-charter.html.

[9] A. Khurri, E. Vorobyeva, and A. Gurtov, "Performance of host identity protocol on lightweight hardware," in *Proceedings of 2nd ACM/IEEE international workshop on Mobility in the evolving internet architecture*, MobiArch '07, (New York, NY, USA), pp. 4:1–4:8, ACM, 2007.

[10] I. Ted Samson, "Amazon EC2 enables brute-force attacks on the cheap." http://infoworld.com/t/data-security/amazon-ec2-enables-brute-force-attacks-the-cheap-447.

[11] A. Lukyanenko, V. Mazalov, A. Gurtov, and I. Falko, "Playing Defense by Offense: Equilibrium in the DoS-attack Problem," in *Proc. of IEEE ISCC'10*, (Piscataway, NJ, USA), IEEE Press, 2010.

[12] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker, "DDoS defense by offense," in *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, (New York, NY, USA), pp. 303–314, ACM, 2006.

[13] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.

[14] V. Krishna, *Auction Theory*. Academic Press, March 2002.