CellFlood: Attacking Tor Onion Routers on the Cheap

Marco V. Barbera[1], Vasileios P. Kemerlis[2], Vasilis Pappas[2], and Angelos D. Keromytis[2]

[1] Sapienza University, Rome, Italy
barbera@di.uniroma1.it
[2] Columbia University, NY, USA
{vpk,vpappas,angelos}@cs.columbia.edu

Abstract. In this paper, we introduce a new Denial-of-Service attack against Tor Onion Routers and we study its feasibility and implications. In particular, we exploit a design flaw in the way Tor software builds virtual circuits and demonstrate that an attacker needs only a fraction of the resources required by a network DoS attack for achieving similar damage. We evaluate the effects of our attack on real Tor routers and we propose an estimation methodology for assessing the resources needed to attack any publicly accessible Tor node. Finally, we present the design and implementation of an effective solution to the problem that relies on cryptographic client puzzles, and we present results from its performance and effectiveness evaluation.

Keywords: Tor network, DoS, client puzzles

1 Introduction

To date, the Tor network [5], one of the most widely used anonymizing systems, consists of more than 3000 Onion Routers that serve daily over 400000 users [25]. Tor helps people all around the world circumvent censorships imposed by oppressive governments, anonymously report abuses of civil rights, and support the freedom of speech and information [28]. It is therefore easy to understand why its security and anonymity properties have attracted a lot of attention over the past years. On the one hand, the community of people and volunteers grown around the Tor network are interested in keeping it secure and operational for its users. On the other hand, however, oppressive governments and organizations may be interested in finding ways to identify people who use it or hinder others from utilizing its services [26].

Being a distributed system operated by volunteers, the anonymity of Tor users is vulnerable to attacks where a set of malicious routers, controlled by an adversary, join the network with the aim of gaining control of user circuits. The Tor network is specifically designed and continuously updated to address these types of threats, but another option available to the adversary would be that of putting a network DoS attack into place with the aim of making it impossible, or very hard, for users to communicate with Tor routers and Tor routers with each other [2]. Such an attack could be used to either significantly degrade the users' perceived quality of service, which would discourage them from using Tor, or to affect the topology of the Tor network in a way that

favours traffic flowing through malicious routers, thus increasing the power of the adversary. A network DoS would not require a deep knowledge of the Tor network internals and could be performed by using well known, pre-existing, off-the-shelf methods [29]. Clearly, since such an attack is orthogonal to those that the Tor network was designed to address, we cannot expect Tor to be resilient to it.

Nevertheless, the protocols used by clients to setup circuits through the Tor network are *vulnerable to a simple attack that would allow an adversary to achieve an effect similar to that of a network DoS, but with just a fraction of its bandwidth resources.* In this paper, we present this attack, named *CellFlood*, and provide an experimental evaluation of it both in a controlled environment and on the real Tor network. Our results, and our estimations based on measurements from a real Tor router, show that CellFlood is not only effective, but also cheap enough to make a feasible alternative to more sophisticated attacks to the Tor network that have been presented in the past. As a way to mitigate the effect of this attack, we propose to use a *client puzzle*-based technique that would allow Tor routers under attack to keep their ability to provide service to honest clients. The main contributions of our work are the following.

- We study CellFlood, a new DoS attack against Onion Routers that significantly impacts their ability to serve circuit creation requests. As opposed to a straightforward network DoS attack, which produces a very large number of lightweight service requests, our attack uses few "heavy" circuit creation requests that can be quickly generated by the attacker on the cheap, but require long processing from the victim. For instance, to halve the processing capability of our least powerful routers, this attack requires only 178 Kb/s, which is the 0.2% of the resources needed for a network DoS attack that matches the maximum Tor data rate supported by the router. For our newest routers, depending on the amount of resources (*i.e.,* CPU cores) they dedicate to serve Tor requests, the attack can require between 2.5 Mb/s (1 core) and 40 Mb/s (16 cores), which is between 1.0% and the 16.0% of the resources needed for an equivalent network DoS.
- We conducted an extensive evaluation regarding the feasibility of the attack both in a controlled environment as well as on the real Tor network. Our findings demonstrate that the attack is effective under different configuration parameters.
- We introduce a lightweight estimation technique for the resilience of a remote, non-cooperative Onion Router to the attack. Using such technique, we identified a set of 48 routers, altogether relaying the 22% of Tor network traffic, that could be attacked with CellFlood using between 2.6 and 9.76 Mb/s of bandwidth per router.
- We discuss the design, implementation, and effectiveness of a mitigation scheme based on client puzzles. Our improved version of the protocol allows routers under attack to easily impose a cap on the attacking host(s), thus preserving their ability to process honest client requests. At the same time, our tests confirm that our protocol has a small impact on the quality of the service perceived by Tor users, even in extreme scenarios.

2 Background

Tor is a distributed overlay network of Onion Routers (ORs), or just routers for brevity, which allows users to get anonymous access to websites and other network services (*e.g.,* SSH, IRC, SMTP, DNS, VNC). Tor decouples clients from the endpoints they aim at connecting to by means of multi-hop paths named *circuits*. Each circuit typically consists of three routers that forward user data from source to destination (and vice versa) in an encrypted way. Data flow through Tor in 512-byte packets, called *cells*, which are routed using "Onion Routing" [11]. When sending data, a Tor client fills the payload of a `RELAY_DATA` cell, and encrypts it iteratively with a different symmetric key (*session key*) for each hop on the circuit. Upon receiving a cell, an OR removes ("peels off") one layer of encryption, with the session key previously negotiated with the client, and forwards the result to the next hop on the circuit (or the final destination).

The negotiation of a session key with each router in a circuit is performed in steps. At each step, the client sends a `RELAY_EXTEND` cell to the latest router OR_i that the circuit has been extended to. The cell wraps an *onionskin*, indicated as $E(g^x)$, where g^x is the first half of a Diffie-Hellman exchange, and E denotes encryption with the *onion key* of the next router OR_{i+1} in the circuit. The onion key is a public 1024-bit RSA key the client previously downloaded from a set of trusted Tor authorities. Upon receiving the `RELAY_EXTEND` cell, OR_i extracts the onionskin and sends it to OR_{i+1} in the payload of a `CREATE` cell. OR_{i+1} uses its private onion key to decrypt g^x, computes the second half g^y of the handshake, a hash of g^{xy}, and sends everything back to OR_i in the payload of a `CREATED` cell. Finally, OR_i forwards the `CREATED` cell to the client by encapsulating it into a `RELAY_EXTENDED` cell.

The procedure to negotiate a session key with the router at the first hop of a circuit is slightly different. Firstly, since there is no other OR between the client and the entry router, the client must put the onionskin directly in the payload of a `CREATE` cell (instead of using a `RELAY_EXTEND` cell). Secondly, most of the times a more lightweight procedure is used that does not involve a Diffie-Hellman exchange, nor public key cryptography. This is because, by default, a Tor client keeps an authenticated and secure TLS connection with a set of three *guard nodes* from which all the circuits are initiated [31]. When using a guard node as the first hop of a circuit, the client and the router exchange the random data used to setup the session key in the payload of a `CREATE_FAST` and `CREATED_FAST` cell, without any further encryption (TLS is sufficient).

3 The CellFlood Attack

Whenever a Tor client extends a circuit, it generates an onionskin using the public onion key of the target router. Likewise, the target router processes it using its private onion key. This operational model makes the processing of onionskins from routers a more expensive than that of generating them. For instance, as we experimentally verified, doing 1024-bit private key operations on a modern high-end server is ~ 20 times slower than doing 1024-bit public key operations [22], which translates into the time to process a CREATE cell being 4 times bigger than that of generating it. This imbalance can be exploited by malicious clients to consume, with relatively small effort, all the computational resources of an OR by means of a continuous stream of `CREATE` cells. To make

matters worse, an attacker does not even have to create a different onionskin for each CREATE cell, as all the cells may contain the same onionskin.

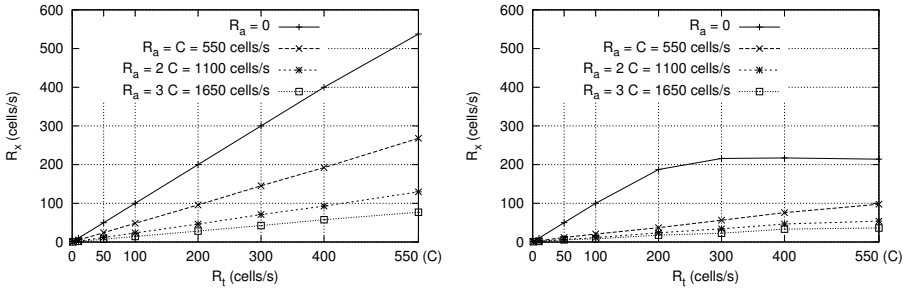
Due to the architecture of Tor software, flooding a Tor router with an excessive number of CREATE requests does not necessarily disrupt the router's ability to forward RELAY_DATA cells. Indeed, Tor delegates the processing of onionskins to a pool of one or more threads (processes), called *CPU Workers*; this allows the main thread (process) to keep up with the more critical work on the RELAY_DATA cells, while the CPU workers perform the expensive and delay-tolerant tasks in the background. Nonetheless, a router that receives CREATE cells at a rate higher from what its CPU workers can process collectively will eventually start discarding them by replying with DESTROY cells. As a consequence, an OR that is under attack is going to discard onionskins produced by honest clients too, which in turn will eventually stop selecting that OR for their circuits. Thus, if CellFlood is performed strategically, on a selected set of important ORs, it may result in overloading the surviving part of Tor (*e.g.,* by overwhelming the unaffected routers with an excessive number or circuits), as well as favouring circuits passing through certain routers, which may well be compromised or controlled by attackers [2] (*i.e.,* similar to the *link-cutting* attack described by Bellovin and Gansner [1]), thus degrading the anonymity of the Tor network as a whole.

According to our experiments, even routers running on recent hardware can only process a limited amount of CREATE cells per second (*i.e.,* a few Mbit/s), which makes them potentially vulnerable to CellFlood. On the other hand, Tor routers can process data to be relayed at a much higher speed, in the order of tens of hundreds of Mbit/s. *Hence, an attacker that is interested in excluding a router or a set of routers from the Tor network will be better off using a stream of CREATE cells, rather than a simpler, but more expensive (in terms of computational resources), DoS attack at the network level.*

3.1 Feasibility Study

Controlled Experiments To study the effect of CellFlood on an OR, we first performed experiments on a controlled environment. In particular, we investigated the capacity of a router, under different attack loads, to process benign CREATE cells that carry onionskins produced by honest clients. Specifically, given the rate R_t cells/s at which legitimate CREATE cells reach the victim router, and rate R_a cells/s at which the attacker sends its bogus CREATE cells, we estimate the final rate $R_x \leq R_t$ of benign cells that can be processed by the router. To launch an attack, we built a custom Tor client that can establish a TLS connection to any victim router in the Tor network and start sending through it a continuous stream of CREATE requests at a specified rate, keeping count of the percentage of requests that get processed. An important aspect of CellFlood is that generating the malicious stream of CREATE cells does not involve any cryptographically heavy operation; all cells have exactly the same onionskin in their payload and differ only on the cell header field storing the id of the circuit to be created.

Our testbed consisted of four hosts connected together through an isolated 100 Mbit/s network. A *Victim* host (armed with a 2.66GHz Core 2 Duo CPU) played the role of the victim OR in a private Tor network. According to our benchmarks, when idle, a single core of *Victim* has a processing *capacity* (C) of ~ 550 CREATE cells/s (~ 2.1 Mbit/s).



(a) No concurrent data traffic. (b) With concurrent data traffic.

Fig. 1. The effect of CellFlood on a Tor Onion Router. R_t denotes the rate of benign cells, R_a is the rate of bogus cells, and R_x shows the cells/s actually processed.

On the other hand, *Victim* can sustain a stream of data cells up to ~ 250 Mbit/s, which shows the magnitude of the advantage an attacker may have in flooding an OR router with `CREATE` cells. The *Attacker* and *Client0* hosts, running on exactly the same hardware (featuring a 3GHz Pentium 4 CPU), are used to generate two concurrent cell streams: one with bogus `CREATE` cells and another with benign. Finally, *Client1* (also armed with a 2.66GHz Core 2 Duo CPU), was used to generate a flow of random data to be forwarded by the victim router. For our experiments, we used Tor v0.2.2.35 with all options set to their default setting; the size of the pending `CREATE` cells queue (`MaxOnionsPending`) and the maximum number of CPU workers (`NumCPUs`) had their default values, 100 and 1, respectively.

Figure 1(a) shows the results obtained when *Victim* processes streams of `CREATE` cells, one from *Attacker*, the other from *Client0*, in absence of any concurrent data stream. Each line in the plot shows how the final rate R_x of accepted client requests varies according to R_t, given a fixed rate of R_a. We varied R_a between C (*i.e.,* 550 cells/sec), $2C$, and $3C$, which corresponds to 2.1, 4.2, and 6.3 Mbit/s of cell traffic. As the figure shows, when there is no attack (topmost line in the plot) all benign onionskins get processed. When the attacker rate R_a matches the capacity C, the number of requests successfully processed drops by approximately a factor of 2, whereas with $R_a = 2C$ and $R_a = 3C$, the drop factor is ~ 4 and ~ 8, respectively.

Next, we evaluated CellFlood under a more realistic scenario, where the victim router processed a stream of `RELAY_DATA` cells coming from *Client1* at the maximum speed allowed by the network, along with the stream of `CREATE` cells. We also configured the victim router to limit its relay bandwidth to 5 MB/s (*i.e.,* 40 Mbit/s), by setting the `BandwidthRate` and `BandwidthBusrst` options, accordingly—this setting is commonly used by Tor routers running on high speed networks for keeping the Tor bandwidth capped. As Fig 1(b) shows, in absence of an attack (topmost line of the plot) the capacity C of the relay dropped to ~ 250 cells/s (~ 0.9 Mbit/s), as opposed to the 550 cells/s that was the original capacity. This is because the stream of `CREATE` cells now competes with the stream of `RELAY_DATA` cells. Nevertheless, the other lines of the plot confirm that CellFlood remains highly successful and has similar effects.

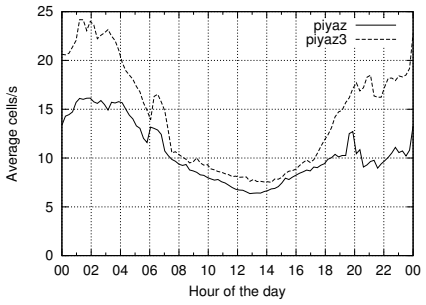


Fig. 2. Daily average of `CREATE` cells/s received by Onion Routers *Piyaz* and *Piyaz3*.

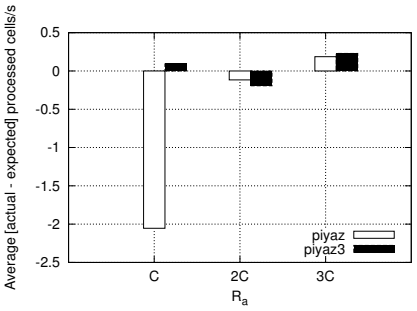


Fig. 3. The accuracy of CellFlood when attacking Tor Onion Routers in the wild.

Experiments in the Wild To assess the effectiveness of CellFlood on public Tor nodes, which are subject to delay, packet loss, *etc.*, we used two ORs under our control that were actively participating to the Tor network. The first, nicknamed *Piyaz* was running on a Xen virtual machine with two virtual 3.06GHz cores, each with a capacity C of 78 cells/s ($\sim$ 0.30 Mbit/s). The other router, nicknamed *Piyaz3*, was a Xeon server with 2.67GHz cores, each with a capacity of 658 cells/s ($\sim$ 2.5 Mbit/s). At the time of the experiments, both *Piyaz* and *Piyaz3* had the `fast`, `stable`, and `guard` flags on, which are given by the Tor authorities to relays with bandwidth and uptime above a certain threshold, so as to provide Tor clients with a hint regarding which routers are the most reliable ones. During our experiments, both ORs were processing an amount of data traffic that varied between 16 and 32 MB/s. Figure 2 shows how the rate of `CREATE` cells/s received by the two routers varies, on average, throughout the day. Both ORs show a similar trend; they receive a higher rate of `CREATE` cells/s at night.

To diversify our tests, we decided to run the CellFlood attack on our routers once every hour, for 2 minutes at a time. Each day, for 3 days, we used a different rate of cells for our attack (*i.e.,* C, $2C$, and $3C$), so as to check whether our results was consistent with those we got from the controlled experiment. The concurrent data traffic that the routers were handling during our tests was always lower than the maximum they were able to process, so we expected our results to be consistent to those shown in Fig. 1(b). Specifically, we expected R_x to be close to $\frac{1}{2}R_t$ for $R_a = C$, $\frac{1}{4}R_t$ for $R_a = 2C$, and $\frac{1}{8}R_t$ for $R_a = 3C$. Our findings are shown in Fig. 3. Each bar represents the average difference between the value of R_x measured during the attack and the value of R_x that we were expecting. The difference was always negligible: when *Piyaz* used as a victim, the difference was $\leq$ 2 cells/s on average, whereas when *Piyaz3* was the victim, the difference was even smaller, always ≤ 0.2 cells/s on average.

4 Global-scale CellFlood

The experiments presented in Sect. 3.1 indicate that an attacker can disrupt the ability of an OR to respond to circuit creation requests, with only a fraction of the bandwidth needed to perform a network DoS of comparable impact. The next step of our study is

to quantify to which extent this is true for core Tor routers. Because of lack of publicly available data regarding the hardware resources of Tor ORs, we remotely measured the capacity of real nodes by means of a custom estimation tool. In the remainder of this section, we will describe the tool and present the results of our estimations.

4.1 Remote Estimation Procedure

We are interested in studying the maximum rate of `CREATE` cells a remote Tor router, not under our control, can process (denoted as C in Sect. 3). This problem is somewhat related to that of estimating the bandwidth capacity of a non-cooperative remote host, for which a number of packet-pair techniques have been proposed in the past [21]. However, these techniques do not fit our purpose. The `CREATE` cells are processed in parallel by multiple CPU workers, and therefore, we have no guarantee that their replies will be received in order. Hence, we opted for a simpler technique that involves flooding the remote router for a short period of time (*e.g.,* one minute) with a train of valid `CREATE` cells, sent at the maximum speed allowed by the network, and counting the percentage F of requests the router was able to process. A value of F less than 1 implies that the router was able to process cells at a smaller rate $R' = R \times F < R$ (recall that R is the rate at which the client sends the `CREATE` cells train), which is a lower bound of the capacity C we try to estimate—this is because during a measurement the OR may receive `CREATE` cells from other clients as well. Thus, by knowing R and F, we can compute R' and use it as an approximation of C. The percentage F can be easily obtained by counting the number of requests replied with a `CREATED` cell over the total number of cells sent, as the number of cells the remote router was not able to process are replied with a `DESTROY` cell[3]. The rate R is simply the number of cells per second at which our client was flooding the target router.

To validate the accuracy of our remote estimation procedure we performed a preliminary test on a small set of 12 ORs that were participating to the Tor network at the time. Thanks to active support from their administrators, we were able to get very accurate estimations regarding the *actual* capacity C of these routers, which could then use as the ground truth for the results obtained through remote estimations. The capacity C was estimated based on: *(i)* the number S of 1024-bit private key operations reported by the OpenSSL `speed` utility, *(ii)* the maximum number of concurrent CPU workers the router is allowed to spawn, and *(iii)* the number of CPU cores available. Specifically, by assuming a linear relation between S and the number N of `CREATE` cells a CPU worker can process per second, it is possible to compute the value N_A of a machine A as $N_A = \frac{S_A N_B}{S_B}$, by just knowing two reference values, S_B and N_B, computed on any other machine B. If the number of CPU cores is at least equal to the maximum number of CPU workers Tor is allowed to spawn, the capacity C of a router A can be then computed as N_A multiplied by the number of CPU workers. Tests performed on a heterogeneous set of machines in our laboratory confirmed that this local estimation technique works within a level of accuracy that is sufficient for our purposes: in all cases, the absolute error in our local estimations was less than 20 cells/s (*i.e.,* $\sim$ 80 Kbit/s), while, on average, the error was about 10 cells/s (*i.e.,* $\sim$ 40 Kbit/s).

[3] Error code: `END_CIRC_REASON_INTERNAL`.

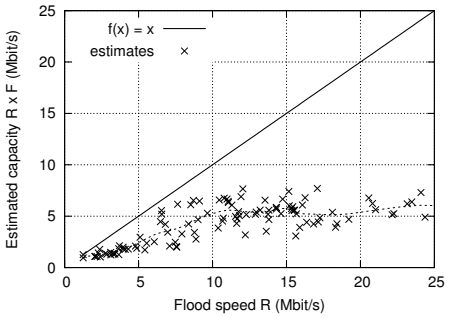


Fig. 4. Example result from the remote capacity estimation. The router was probed every 2 hours, for 60 seconds, over a period of 11 days.

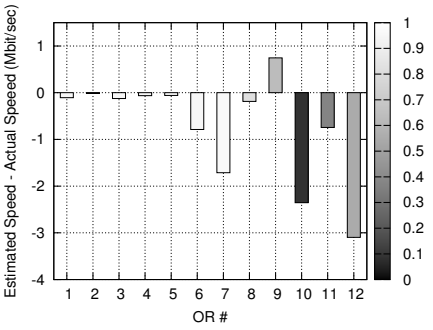


Fig. 5. Accuracy of the remote capacity estimation procedure. The shade represents the confidence of the measurement (lighter shade correspond to higher confidence).

With the results of our local estimation at hand, we remotely estimated the capacity of the routers in the test set during a timespan of 11 days. The estimation of the capacity of a router is the result of 126 measurements (one every 2 hours) each lasting 60 seconds. Each measurement produces a pair of values (R, F), where R is the speed at which the cells were reaching the router and the product $R \times F$ is the estimated capacity. Figure 4 shows the measurements relative to a router in the test set. Each point in the plot represents a measurement, and the line $f(x) = x$ is where the points relative to the measurement that did not hit the capacity of the router would lie. The figure shows a clear trend: as the the rate R increases, the value $R \times F$ starts following roughly the $f(x) = 7$ line, meaning that the measurements hit the capacity of the router and forced it to discard some cells.

For each router our measurements yielded two values: the maximum estimated capacity, and a confidence metric for the accuracy of the estimation. The former is given by the point $\max_i (R_i, F_i)$—*i.e.,* the measurement that maximizes the estimated capacity $R_i \times F_i$. The latter is the value $1.0 - F_i$, that is, the percentage of cells the router was not able to process during the measurement that produced the highest estimated capacity. The intuition behind this choice is that, the higher the percentage $1.0 - F_i$ of cells the router was forced to discard, the more certain we can be about the rate R_i having exceeded the capacity of the router—*i.e.,* the value $R_i \times F_i$ is a good approximation of the capacity. Although the confidence metric gets values in the interval $[0, 1]$, values close are 1 are not common. For instance, a confidence of 0.9 would mean that the rate R was ten times bigger than the capacity of the router, which can presumably happen only when the capacity of the router is very low. We thus deem confidence values of at least 0.5 as high, since they are produced by measurements where the rate R was at least twice as the capacity.

Figure 5 shows the aggregate results of the estimations that we performed on the 12 ORs. Each bar represents the difference between the remotely and locally estimated capacity of a given router. The shade of the columns represents the confidence metric described above. The lighter the shade, the higher the confidence. As the picture shows,

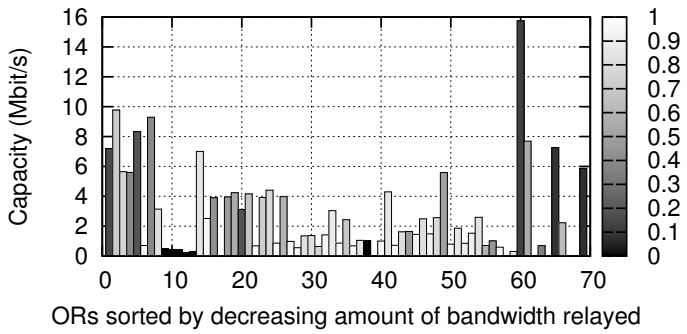


Fig. 6. Estimated capacity C for the 78 Tor Onion Routers.

in almost all the cases our remote estimation technique gives a lower bound of the actual capacity of the routers (*i.e.,* the columns have negative values). This is normal, since the routers were probably receiving a concurrent flow of `CREATE` cells during our measurements. The only case were we overestimated the capacity was for router number 9. However, this is acceptable for the type of study we are doing (*i.e.,* the router is computationally "weaker" from what we think).

4.2 Estimating the Effects of CellFlood on Tor

We ran our remote estimation tool on a set of 78 routers selected according to both the amount of time they had been part of the network (at least 2 months) and the amount of data they were relaying, as it was reported in [27]. This allowed us to get a snapshot of the routers that were part of the "core" of Tor, that is, the set of ORs on which the network was depending on in order to provide a good and reliable service to its users. At the time of the experiment (May 2012), these routers were responsible for the $\sim 50\%$ of the total traffic flowing through the Tor network. As with the tests of Sect. 4.1, we ran our remote estimation tool every 2 hours for 11 days, with each measurement lasting 60 seconds. Results are shown in Fig. 6. The height of each bar in the plot represents the estimated capacity C of a router, whereas the shade represents the confidence of the measurement (*i.e.,* the lighter the bar, the higher the confidence). For some of the routers, especially those in the range 14–59, we measured low C values (around 2 and 4 Mbit/s) with high confidence (above 0.5). On the other hand, the measurements of the capacity of the topmost 8 routers yielded higher values (4 to 10 Mbit/s), but for two of them the confidence was rather low (less than 0.4). The measurements of the 5 routers in the range 9–13 yielded very low capacity (less than 1 Mbit/s) but a confidence level very close to 0, meaning that there was never enough bandwidth between them and our measuring machines to give an accurate estimation. The router number 60 was the one with highest capacity (close to 16 Mbit/s), plus, the confidence of the measurement was close to 0.1, so we can expect the actual capacity of that router to be even higher. In general, our confidence values were high (*i.e.,* at least 0.5) for the 62% of the routers, which were relaying for 22% (*i.e.,* 2.8 Gbit/s) of the total amount of

data flowing through the Tor network at that time. Considering our findings in Sect. 3.1, these values are low enough to open the possibility for an attacker to cause a significant disruption to specific routers or to the Tor network as a whole with relatively small bandwidth resources. For instance, the total bandwidth needed in order to flood those 62% of the routers whose capacity was measured with high confidence would just be around 116 Mbit/s. Even in the pessimistic case where our estimation gave only the 50% of the actual values (which, given our high confidence values, is unlikely) the total bandwidth needed by an adversary to clog them would just be 232 Mbit/s.

5 Client Puzzles to the Rescue

As a countermeasure to the CellFlood attack we propose a solution based on *client puzzles*. With client puzzles, a server under attack commits the resources needed to satisfy a given request (*i.e.,* processing an onionskin) only after the client has performed some computationally intensive work, usually in the form of solving a cryptographic problem. This adds a computational constraint to the attacking host(s), thus reducing the power of the adversary. Client puzzles are a good fit for Tor for several reasons. First, each router can defend itself against a CellFlood attack without cooperating with other ORs. This is consistent with the trust model of Tor, where any router can turn out to be malicious. Second, client puzzles are not affected from how the attack is orchestrated. That is, whether coming directly from a router or a client, or indirectly, through another router by encapsulating onionskins into `RELAY_EXTEND` cells (instead of `CREATE` cells). Third, the topology of the network will be preserved, as routers under attack will not be forced to close any active connections in the hope of stopping the attack. Finally, the difficulty of client puzzles can be adjusted according to the strength of the attack, thus making the solution effective even in the case of a global-scale CellFlood.

Figure 7 shows how the client puzzle protocol works when establishing a session key with the first two hops of a circuit. The procedure for the 3$^{\text{rd}}$ (4$^{\text{th}}$, 5$^{\text{th}}$, *etc.*) hop is similar and omitted for brevity. If a `CREATE_FAST` cell is used for the first hop of the circuit, puzzles will never be issued, as no intensive cryptographic operation is required (see Sect. 2). In the remainder of this section, we will discuss in great detail the design and implementation of our mitigation scheme for CellFlood attacks that is based on client puzzles.

5.1 Building and Solving Puzzles

Our client puzzles are built upon SHA256-based message authentication codes (HMAC). To build a puzzle, the router generates s (a random 64-bit key) and computes the value $X = \text{HMAC}(s, P|H)$, where $P|H$ is a message resulting from the concatenation of the onionskin P contained in the payload of the `CREATE` cell and the hash H of the router's public *identity key*, which is a long-term public key that establishes the router's identity. Finally, a key s' is generated by setting the least k-bits of s to 0. The puzzle is the triplet (s', k, X) and does not include the HMAC message $P|H$, since P and H are both known to the client. To solve the puzzle, the client has to guess the k unknown bits

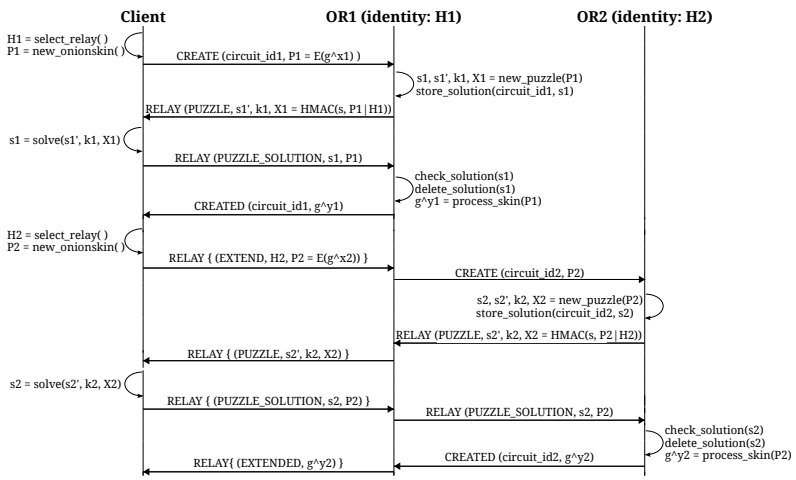


Fig. 7. Client puzzle protocol for mitigating CellFlood attacks (assuming that `CREATE_FAST` cells are not used and puzzles are send at each hop). '{}' denotes encryption with the session key.

of s starting from s', by computing, for each tentative s'', HMAC$(s'', P|H)$ and comparing it against X to check whether $s'' = s$. Since finding a pre-image for SHA256 is computationally infeasible, a puzzle with k unknown bits requires an average of 2^{k-1} tentatives, which grows exponentially with k. This allows the puzzle complexity to be adjusted at will from few milliseconds up to several hours.

5.2 Sending Puzzles and Solutions

As Fig. 7 shows, puzzles and puzzle solutions travel in the payload of `RELAY` cells with command code `PUZZLE` and `PUZZLE_SOLUTION`, respectively. These cells are subsequently encrypted (resp. decrypted) with the session key previously negotiated with any router in the portion of the circuit built thus far. The only router that can read both the puzzle and its solution is the last one, which stands between the client and the router that issued the puzzle. That is, OR_1 in Fig. 7, when the client is extending the circuit to OR_2. What prevents OR_1 to maliciously interfere with the protocol is the fact that the client expects the HMAC message used to generate the puzzle to be $P_2|H_2$. This ensures that the client will refuse to solve any puzzle produced with another pair of P_2 and H_2, and that no other router but OR_2 (with identity H_2) will accept the solution.

Another important detail is that the use of `RELAY` cells (to send puzzles and puzzle solutions) makes our protocol backwards compatible with the ORs that implement the original Tor protocol, which does not support puzzle messages. These routers will be able to encrypt/decrypt `PUZZLE` and `PUZZLE_SOLUTION` cells, similarly to any other `RELAY` cell. Also, a non puzzle-compatible client that receives a `PUZZLE` cell will ignore it and try a different OR. This allows for the incremental adoption of our solution, since it does not require all routers and clients to upgrade their software at once.

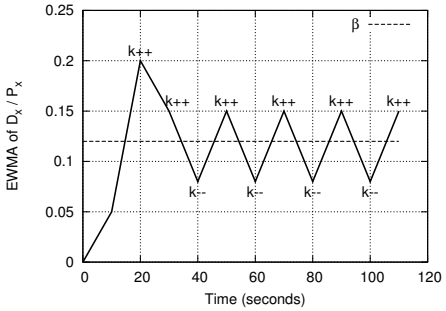


Fig. 8. The change of parameter k when the Onion Router is under attack (Δ_x set to 10s, β set to 0.12).

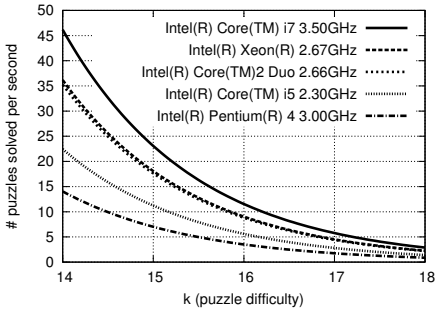


Fig. 9. Time required to solve puzzles of varying difficulty.

5.3 Verifying Solutions

To check whether a puzzle solution is correct, a router compares the received solution s to the one stored when the puzzle was generated (`check_solution` in Fig. 7). In our implementation, storing a puzzle solution requires 18 bytes of memory (8 for the puzzle solution + 8 for the connection id + 2 for the circuit ID), and it stays in memory until the router receives a reply from the client or a timeout Δ_p expires (`delete_solution`). The role of the timeout is to prevent an attacker from consuming all the memory of a router, by leaving puzzles unsolved.

Choosing a good value for Δ_p is not hard; even with a Δ_p as big as 2 minutes, an attack of 189 Mbit/s will consume 100 MB of memory. Considering that our measurements have shown that the most important routers of the Tor network can support a stream of `CREATE` cells of a few Mbit/s (see Sect. 4.1), an attacker that dedicates 189 Mbit/s of bandwidth for clogging *each* router is much more powerful than the model of the adversary we are considering. Also, according to our tests, a timeout of 2 minutes is way more than enough even for slow clients to be able to solve the puzzle on time. Nevertheless, the Δ_p parameter can be adjusted to greater or smaller values depending on the situation. An alternative strategy could be to not store the solution, and give each puzzle an expire time to avoid an adversary reusing the same solution multiple times. Setting the expire time, however, is a complicated task that requires carefully estimating the capacity of the attacker (in terms of bandwidth) and the speed at which honest clients can solve puzzles. A big value will give too much power to the attacker, whereas a small value will discriminate slow honest clients. We believe that our solution strikes a balance between security and performance, as it comes with moderate cost.

5.4 Choosing the Puzzle Difficulty

Two possible approaches can be used here: *(i)* always send puzzles to clients, or *(ii)* send puzzles only when an attack is in place. The first approach is simpler, but most of the time it imposes an unnecessary load on Tor clients. The second one is lightweight, but it

involves inferring whether a DoS attack is in place or not. For the latter case, a custom and lightweight approach could be used where each Δ_x seconds the router counts the total number P_x of CREATE cells that it was able to process and the number D_x of those that it had to discard because all the CPU workers were busy. An exponential moving average (EWMA) $\overline{\mu} \in [0, \inf)$ of the fraction D_x/P_x could used to detect when the average percentage of dropped cells reaches a threshold value β. The first time $\overline{\mu}$ becomes bigger than a threshold β, the router starts sending puzzles with a initial difficulty parameter k (*e.g.,* $k = 16$) for Δ_x seconds. At the end of the interval, the router increases or decreases the difficulty k of the puzzle by one bit depending on whether the updated value of $\overline{\mu}$ has become smaller than β or not. This would allow the router to continuously adjust the difficulty parameter k, as shown in Fig. 8. To avoid imposing a too heavy load on an honest client, the maximum puzzle difficulty could be set up to be around 20. The parameter β should be set to very low values (even 0), depending on how likely the router administrator believes his CPU workers will be discarding cells during the normal operation time. Finally, the value of Δ_x is not very critical, but it should be short enough to allow quickly finding the right puzzle difficulty.

5.5 Testing and Evaluation

Puzzle Solution Performance We studied the time it takes to solve client puzzles on a wide range of machines, which vary from slow, outdated hardware, to brand-new, high-end workstations. Figure 9 shows the speed at which our machines were able to solve puzzles for $k \in [14 - 18]$. As the figure shows, even for a value as small as 17, our fastest host (armed with an Intel Core i7 3.5GHz CPU) can solve just around 6 puzzles per second. It is interesting to notice that for a fixed k, there is no big difference between the performance of different hardware. For instance, our slowest machine (the 3GHz Pentium 4) is slower by a factor of ~ 3.2 with respect to our fastest one. Thus, a router can easily estimate, within a reasonable level of accuracy, what would be the impact of sending a puzzle of a difficulty k to honest clients and attackers alike. According to this data, using a puzzle of difficulty $k = 18$ should be good enough for slowing down an attack performed with today's off-the-shelf hardware. For instance, if the capacity C of a victim router is 300 cells/s, which is lower than that of *Piyaz3*, an attacker should use the equivalent of around ~ 100 cores of our fastest Core i7 3.5GHz machine, to successfully clog the router. On the other hand, the slowest clients should only experience a delay of around 1 second.

Puzzle Generation Performance To evaluate the load imposed by the client puzzle protocol on ORs, we studied the time it takes for our test machines to read a `CREATE` cell and generate a puzzle, and the time it takes to read a `PUZZLE_SOLUTION` cell and verify the solution. The tests machines used were *Victim* and *Client1*, plus the two routers *Piyaz* and *Piyaz3* (see Sect. 3.1). Figure 10 compares the capacity of these machines to process `RELAY` cells containing data to be forwarded (`RELAY_DATA` column), to read a `CREATE` cell and generate the relative puzzle (`RELAY PUZZLE` column), and to check the solution (`RELAY PUZZLE SOLUTION` column). For clarity, the capacity is shown in Mbit/s instead of cells/s, knowing that a cell size is 512 bytes. As the figure

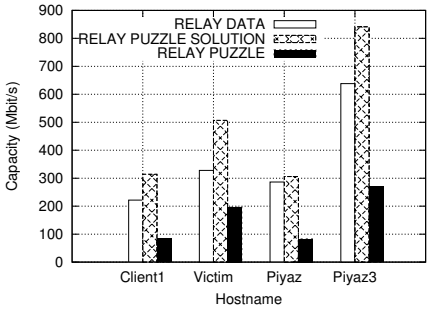


Fig. 10. Time required by different Onion Routers to generate puzzle cells and verify puzzle solutions, compared with the time it takes to process regular RELAY_DATA cells.

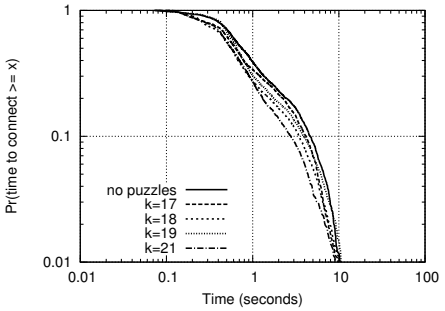


Fig. 11. Distribution of time-to-connect when simulating a Tor client solving client puzzles.

shows, even our slowest machines, namely *Client1* and *Piyaz*, when idle, can process CREATE cells and generate the relative puzzles at around 90 Mbit/s. For *Victim* and *Piyaz3*, the value is at least 200 Mbit/s. On the other hand, checking whether the puzzle solution is correct is much faster, even faster than processing a RELAY_DATA cell, since it just requires comparing the received solution with the one the router stored in memory. From the security point of view, the values we got for the puzzle generation on *Victim* and *Piyaz3* do not represent an issue. An adversary that floods a router with 200 Mbit/s of data represents a threat anyway. For what concerns the slower machines *Client1* and *Piyaz*, performances can be improved using a SHA1-based HMAC (instead of the SHA256-based HMAC). According to our experiments, this would increase their capacity to generate puzzles up to 230 Mbit/s. To compensate for the higher speed at which clients would be able to solve SHA1 HMAC puzzles, the routers should slightly increase the difficulty k of their puzzles by one or two bits. Obviously, using a different parameter k does not affect the puzzle generation speed in a noticeable way.

Quality of Service Finally, we considered how the user-perceived quality of service would change in case our puzzle protocol was actually used in the Tor network. To evaluate it, we set up an automatic test where a modified version of the Tor client introduced fake delays in the generation of circuits. This allowed us to simulate both the time delay caused by the transmission of puzzles and puzzle solutions, and the time spent by the client to solve the puzzles. The first type of delay is computed starting from the Δ_1, Δ_2, and Δ_3 intervals of time needed for the RELAY_EXTEND and RELAY_EXTENDED (or CREATE and CREATED) cells to be exchanged between the client and the routers at the first, second, and third hops of the circuits respectively. By adding Δ_i to the circuit creation time, we simulate the exchange of the PUZZLE and PUZZLE_SOLUTION cells between the client and the i-th hop of the circuit. Note that if the client uses a CREATE_FAST cell for the first hop of the circuit, puzzles will not be used and the extra delay Δ_1 will be ignored. The simulation of the time needed to solve each puzzle was based on a parameter telling the difficulty k of the puzzles. Again, this delay is

Difficulty k	Avg. Sol. Time (seconds)	CPU Idle time %	Avg. CPU usage % (when not idle)
0	0	90	3
17	0.5	90	23
18	1.0	87	37
19	2.0	85	52
21	8.0	66	75

Table 1. CPU usage when simulating a Tor client solving client puzzles.

added to each hop of the circuit except for the first one, unless the client decided to use a `CREATE` cell instead of a `CREATE_FAST` cell.

To automate the test, we implemented a simple HTTP client consecutively fetching 500 random web pages through our Tor client. The client randomly pauses between each request and the next, so as to simulate the time between user's clicks. The length of the pause is drawn from the UNC *think-time* distribution [12]. This distribution is also used by Jansen and Hopper [13] when simulating Tor users activity in their Shadow simulator. In our evaluation, we focused only on the time it takes for the HTTP client to connect to the web server hosting the page, as our client puzzle protocol affects solely the creation of the circuits. Results are shown on Fig. 11 and Table 1. In these experiments we assumed both the best-case scenario when no router asks the client to solve puzzles (*i.e.,* Δ_i is not used), and the worst-case scenario where *all* the routers send puzzle cells in reply to the client's `CREATE` cells (*i.e.,* both Δ_2 and Δ_3 are always used, whereas Δ_1 is not used in case of a `CREATE_FAST` cell). In the latter case, the puzzle difficulty parameter k was set to 17, 18, 19 and 21. The results are computed by running 5 independent tests for each value k. The Tor client was running on *Client1*, which is also the oldest machine we had available in our lab.

6 Related Work

Most of the research on Tor has focused on techniques aimed at degrading user's anonymity by means of congestion attacks (Evans *et al.* [7], Murdoch and Danezis [19]), web page fingerprints (Shi and Matsura [23]), observations of the throughput of Tor streams (Mittal *et al.* [16]), or by means of colluding nodes (Fu *et al.* [9], Levine *et al.* [14]). Other attacks study the potential threat of a (semi) *global* adversary (Murdoch and Zieliński [17], Edman and Syverson [6], Chakravarty *et al.* [3]), although this does not fit into Tor's original adversary model. Specific attacks to Tor bridges and Hidden Services have been studied by McLachlan and Hopper [15], and by Murdoch [18], respectively. Borisov *et al.* [2] study a selective DoS attack where Tor routers controlled by an adversary relay only messages of circuits they can fully deanonymize, by controlling the first and the last hop, while disrupting everything else. They show that this type of adversary has a significant advantage over a passive adversary like the one presented by Syverson *et al.* [24]. Danner *et al.* [4] give a countermeasure to this attack that works by probing the network for misbehaving routers. The proposed technique is able to detect all the adversary-controlled routers with $O(n)$ probes, where n is the total number of routers of the Tor network. A DoS-like packet spinning technique is presented by

Pappas *et al.* [20]. By increasing the circuit creation latency of the honest routers, they allow an adversary to increase the probability of malicious routers to be selected. As the method presented by Evans *et al.* [7], this attack works by building arbitrary long circuits, which has become harder since Tor v0.2.1.3.

To the best of our knowledge, the only work in literature that is close to ours is the one presented by Fraser *et al.* [8]. Their attack, however, exploits the well known DoS attack against the TLS handshakes, whereas we focus on the circuit creation protocol, which is specific to Tor only. Their solution is based on stateless client puzzles, but they do not evaluate the impact of the time-window parameter telling how long a puzzle solution is valid, which, on the other hand, we believe is critical (more about this in Sect. 5.3). Also, as opposite to us, they do not give an estimation of the vulnerability of the routers currently being part of the Tor network. Finally, their solution might actually make it easier for censoring devices to spot Tor bridges by means of fingerprint attacks (interested readers are referred to the recent study by Winter and Lindskog [30]).

7 Conclusions

In this paper, CellFlood, a DoS attack that exploits a security weakness in the circuit creation protocol of the Onion Routers has been evaluated for the first time. Our results, based on tests in a controlled environment and on an estimation performed on a set of crucial Tor nodes, have confirmed that this attack could be not only possible but also effective, and easier to perform, than a standard network DoS. We have proposed, fully implemented, and evaluated a backward-compatible solution based on a client puzzle protocol that would allow Tor nodes to increase at will the computational resources needed to perform this attack. Our results show that the load imposed to honest clients by our improved protocol would be moderate even in a worst-case scenario.

Acknowledgments This work was supported by DARPA and the NSF through Contracts FA8650-11-C-7190 and CNS-12-22748, respectively. Any opinions, findings, conclusions or recommendations expressed herein are those of the authors, and do not necessarily reflect those of the US Government, DARPA, or the NSF.

References

1. Bellovin, S.M., Gansner, E.R.: Using link cuts to attack Internet routing. Tech. rep. (2002)
2. Borisov, N., Danezis, G., Mittal, P., Tabriz, P.: Denial of service or denial of security? In: CCS. ACM (2007)
3. Chakravarty, S., Stavrou, A., Keromytis, A.: Traffic analysis against low-latency anonymity networks using available bandwidth estimation. Computer Security–ESORICS (2010)
4. Danner, N., Krizanc, D., Liberatore, M.: Detecting denial of service attacks in tor. Financial Cryptography (2009)
5. Dingledine, R., Mathewson, N., Syverson, P.: Tor: The second-generation onion router. Tech. rep., DTIC Document (2004)
6. Edman, M., Syverson, P.: As-awareness in tor path selection. In: CCS. ACM (2009)

7. Evans, N., Dingledine, R., Grothoff, C.: A practical congestion attack on tor using long paths. In: USENIX Security. USENIX (2009)
8. Fraser, N., Kelly, D., Raines, R., Baldwin, R., Mullins, B.: Using client puzzles to mitigate distributed denial of service attacks in the tor anonymous routing environment. In: ICC. IEEE (2007)
9. Fu, X., Ling, Z., Luo, J., Yu, W., Jia, W., Zhao, W.: One cell is enough to break tors anonymity. In: Black Hat Technical Security Conference (2009)
10. Goldberg, I., Stebila, D., Ustaoglu, B.: Anonymity and one-way authentication in key exchange protocols. Designs, Codes and Cryptography (2012)
11. Goldschlag, D., Reed, M., Syverson, P.: Hiding routing information. In: Information Hiding. Springer (1996)
12. Hernández-Campos, F., Jeffay, K., Smith, F.: Tracking the evolution of web traffic: 1995-2003. In: MASCOTS. IEEE (2003)
13. Jansen, R., Hopper, N.: Shadow: Running tor in a box for accurate and efficient experimentation. Tech. rep., DTIC Document (2011)
14. Levine, B., Reiter, M., Wang, C., Wright, M.: Timing attacks in low-latency mix systems. In: Financial Cryptography. Springer (2004)
15. McLachlan, J., Hopper, N.: On the risks of serving whenever you surf: vulnerabilities in tor's blocking resistance design. In: Workshop on Privacy in the electronic society. ACM (2009)
16. Mittal, P., Khurshid, A., Juen, J., Caesar, M., Borisov, N.: Stealthy traffic analysis of low-latency anonymous communication using throughput fingerprinting. In: CCS. ACM (2011)
17. Murdoch, S., Zieliński, P.: Sampled traffic analysis by internet-exchange-level adversaries. In: Privacy Enhancing Technologies. Springer (2007)
18. Murdoch, S.: Hot or not: Revealing hidden services by their clock skew. In: CCS. ACM (2006)
19. Murdoch, S., Danezis, G.: Low-cost traffic analysis of tor. In: Security and Privacy. IEEE (2005)
20. Pappas, V., Athanasopoulos, E., Ioannidis, S., Markatos, E.: Compromising anonymity using packet spinning. Information Security (2008)
21. Prasad, R., Dovrolis, C., Murray, M., Claffy, K.: Bandwidth estimation: metrics, measurement techniques, and tools. Network, IEEE 17(6) (2003)
22. RSA Laboratories: How fast is the RSA algorithm? https://www.rsa.com/rsalabs/node.asp?id=2215.
23. Shi, Y., Matsuura, K.: Fingerprinting attack on the tor anonymity system. Information and Communications Security (2009)
24. Syverson, P., Tsudik, G., Reed, M., Landwehr, C.: Towards an analysis of onion routing security. In: Designing Privacy Enhancing Technologies. Springer (2001)
25. Tor Project: Tor metrics portal. https://metrics.torproject.org
26. Tor Project: Using tor hidden services for good. https://blog.torproject.org/blog/using-tor-good
27. TorStatus: http://torstatus.blutmagie.de/.
28. WikiLeaks: Tor. http://www.wikileaks.org/wiki/WikiLeaks:Tor
29. Wikipedia: Low orbit ion cannon. http://en.wikipedia.org/wiki/Low_Orbit_Ion_Cannon
30. Winter, P., Lindskog, S.: How china is blocking tor. arXiv preprint arXiv:1204.0447 (2012)
31. Wright, M., Adler, M., Levine, B., Shields, C.: Defending anonymous communications against passive logging attacks. In: Security and Privacy. IEEE (2003)

A User-perceived Quality of Service

In Table 1, the difficulty parameter k of the puzzles is compared to the average time needed to solve a puzzle (second column), to the percentage of time that the CPU was idle during the tests (third column), and to the average CPU usage level when the CPU was *not* idle (last column). From the table it can be observed that, as the difficulty of the puzzle increases, the average CPU idle time decreases and the average CPU usage percentage increases. It is interesting to see that although the time it takes to solve a puzzle is as high as 8 seconds, the average CPU load is only 75%. Figure 11 represents the distribution of the time-to-connect for varying client puzzle difficulty. As the figure shows, there is never a relevant difference in the time it takes for the connection to be established. In other words, the user's perceived quality of service is not affected in a noticeable way, not even in the case of $k = 21$ and all routers requiring the client to solve puzzles before building a circuit. There reason is that the Tor client software maintains a small pool of pre-built circuits that can readily serve new user request. Plus, "dirty" circuits are reused for a certain amount of time before being closed definitively. This design choice has been done in order to deal with any network delay there might be in the creation of circuits. As our experiments have shown, this mechanism is good enough to absorb the extra-delay imposed by client puzzles too.

One last important detail is that of our custom implementation of Tor delegates the solution of the received puzzles to a pull of CPU workers, so as to avoid introducing a delay in the processing of data to/from the Tor network. Using a single CPU worker was sufficient even on the slow machine we used for these tests.

B `ntor` handshake

Starting with v0.2.4.8-alpha (released in January 2013), Tor supports a new circuit extension handshake protocol, `ntor`, designed by Goldberg *et al.* [10]. `ntor` improves upon the original protocol we described in Sect. 2, both in terms of security and speed, by using Dan Bernstein's "curve25519" elliptic-curve Diffie-Hellman function. As of June 2013, only about 7% of the Tor routers support this new handshake protocol, although this percentage is destined to grow over time. Preliminary tests performed in our laboratory testbed confirm that, depending on the machine, `ntor` provides a speed up factor (in processing create circuit requests) of up to 4x. Nevertheless, as with the original circuit extension handshake protocol, `ntor` maintains the imbalance in the amount of resources needed for clients and routers to extend a circuit, and as such, it remains vulnerable to CellFlood.