Revisiting Client Puzzles for State Exhaustion Attacks Resilience

Mohammad A. Noureddine*, Ahmed M. Fawaz†, Amanda Hsu†, Cody Guldner*
Sameer Vijay*, Tamer Başar†, William H. Sanders†
*Department of Computer Science, †Department of Electrical and Computer Engineering
University of Illinois at Urbana Champaign

***Abstract*—In this paper, we address the challenges facing the adoption of client puzzles as a means to protect the TCP connection establishment channel from state exhaustion DDoS attacks. We model the problem of selecting the puzzle difficulties as a Stackelberg game with the server as the leader and the clients as the followers and obtain the equilibrium solution for the puzzle difficulty. We then present an implementation of client puzzles inside the TCP stack of the Linux 4.13.0 kernel. We evaluate the performance of our implementation and the obtained solution against a range of attacks through reproducible experiments on the DETER testbed. Our results show that client puzzles are effective at boosting the tolerance of the TCP handshake channel to state exhaustion DDoS attacks by rate limiting malicious attackers while allocating resources for legitimate clients.**

***Index Terms*—Denial of Service Attacks, Proof-of-Work, Stackelberg Games, Transport Control Protocol**

I. Introduction

In recent years, the scale and complexity of Distributed Denial of Service (DDoS) attacks have grown significantly. The introduction of DDoS-for-hire services has substantially decreased the cost of launching complex, multi-vectored attacks aimed at saturating the bandwidth as well as the state of a victim server [1], [2]. Common mitigations for large-scale DDoS attacks are focused around cloud-based protection-as-a-service providers, such as CloudFlare. When under attack, a victim's traffic is redirected to massively over-provisioned servers, where proprietary traffic-filtering techniques are applied and only traffic deemed benign is forwarded to the victim. The relative success of such over-provisioning techniques in absorbing *volumetric* attacks has pushed attackers to expand their arsenal of attacks to span multiple layers of the OSI network stack [3]. In fact, 39.8% of the attacks launched through the Mirai botnet were aimed at TCP state exhaustion, while 32.8% were volumetric [4]; the source code contained more than 10 vectors in its arsenal of attacks [5].

State exhaustion attacks are particularly challenging as they target *stateful* devices such as firewalls, load balancers, and application servers. Attackers can disguise them as benign traffic by leveraging a large number of machines that can use their authentic IP addresses [1], and can bypass cloud-based protection services [6], [7], capabilities, and filtering techniques [8]–[13] by sending slower and non-spoofed traffic. This situation is further exacerbated by the imbalance between the cost of launching a multi-vectored DDoS attack and the cost of mitigating one. Launching an attack incurs an average cost of $\$66$ per attack and can cause damage to the victim of around $\$500$ per minute [14]. Furthermore, such attacks cannot be mitigated by employing geo-distributed *Content Delivery Networks* (CDN) as these CDNs still need to provide stateful services (such as TCP and HTTP) to the original victim's users; cloud protection services in fact recommend that defenses against state exhaustion attacks should reside at the victim's premises [6].

In this paper, we revisit the application of client puzzles as a mechanism for resisting state exhaustion DDoS attacks. Client puzzles are a promising technique that alleviates the cost imbalance between the attacker and the defender with only software-level modifications at the end hosts and no changes to the Internet infrastructure [15], [16]. In essence, client puzzles attempt to hinder the malicious actors' ability to flood the server by forcing all clients, benign and malicious, to solve computational puzzles for each request they make.

While TCP client puzzles are a promising technique for resisting state exhaustion attacks, they have not seen their way into adoption because of (1) the lack of guidelines on how to set the difficulty, and (2) the lack of publicly available implementations and performance studies [15], [17], [18]. A TCP client puzzle's difficulty determines the computational burden placed on the server and clients. Current standards [18], [19] suggest using a fixed difficulty level for all clients in order to maintain a stateless protocol, they however do not provide any sound ways for selecting the appropriate difficulty level. Difficulty selection becomes even more challenging when the victim serves clients with a mixture of power-endowed devices. Additionally, the few existing implementations of TCP client puzzles [17], [20]–[22] are outdated and are not publicly available, further hindering the community's ability to evaluate their effectiveness and adopt them.

In this work, we make the following contributions to address the shortcomings of TCP client puzzles research. First, we introduce a theory for determining an appropriate TCP puzzle difficulty based on the game-theoretic Stackelberg interaction between the defender and the clients [23]–[25] (Sections III and IV). Using the theory we established, we provide a practical method for selecting the TCP puzzles difficulty based on the defender's capabilities and the expected computational prowess of the clients.

Then, we describe how we designed, implemented and evaluated an extension to TCP to support client puzzles using

our practical difficulty-setting method. We incorporate puzzles into the TCP handshake and otherwise do not interfere with the operation of the protocol. We efficiently encode the challenges and their solutions into the *options* of the TCP header, resulting in low packet-size overhead. Then, we implement TCP puzzles as part of the Linux kernel TCP stack (Section V). Our patch is publicly available at `https://github.com/nouredd2/puzzles-utils`.

We evaluated the performance of our TCP puzzles against a range of attacks through reproducible experiments performed using the DETER testbed (Section VI). Our results show the effectiveness of TCP puzzles in boosting tolerance against state exhaustion attacks. If a server using client puzzles with our game-theory-based difficulty setting method, it can tolerate both SYN and connection floods that would bring down an unprotected server or one that relies solely on SYN cookies [26]. Finally, we present a preliminary study of *puzzles-based queueing* (Section VI-E) as a technique to provide fair treatment to a mixture of power-endowed devices.

II. Background and Related Work

In this section, we review the TCP three-way handshake and TCP state exhaustion attacks. We then present client puzzles and their current limitations. For the remainder of this paper, we use the terms *puzzles* and *challenges* interchangeably.

A. TCP primer and SYN flood attacks

In current TCP implementations, a client initiates a TCP connection by sending a SYN packet to the server. Upon receiving the SYN packet, the server saves state for this new incoming connection request in a data structure, often referred to as the *Transmission Control Block* (TCB), and then sends a SYN-ACK packet and waits for the client to acknowledge receipt of this packet. A half-open connection is one for which the client's ACK packet has not yet been received; those new connection sockets are queued into a `listen` queue. The number of elements in this queue is upper-bounded by an implementation parameter, called the *backlog*, that bounds the server's memory usage to avoid exhaustion of the system's resources. Once a connection has been established, the server moves it into the `accept` queue. A socket is removed from the accept queue once the server's application accepts it for processing. On the other hand, a half-open connection socket is removed from the queue if it expires before it receives an acknowledgment from the client [27]. Once the server's listen or accept queue overflows, it either (1) no longer accepts incoming connections, or (2) drops old connection sockets from the appropriate queue.

TCP SYN flood attacks aim to overflow a victim server's `listen` queue by overwhelming it with half-open connection requests. The attack forces the server to drop new incoming connections, thus denying service to new clients [28]. A variant of the TCP SYN flood attack is a TCP connection flood in which an attacker attempts to overflow the server's `accept` queue for the same purpose of denying legitimate clients the opportunity to connect to the server. In a connection flood, the attacker completes the three-way handshake instead of leaving the connections half-open.

Among the server-based mitigations for SYN flood attacks, the SYN cache and TCP SYN cookies are the most common [26], [28], [29]. The SYN cache reduces the amount of memory needed to store state for a half-open connection by delaying the allocation of the full TCB state until the connection has been established. Servers that implement SYN caches instead maintain a hash table for half-open connections that contains partial state information and provides fast lookup and insertion functions. SYN cookies, on the other hand, operate by eliminating the source of the vulnerability in TCP implementations: the state reserved for half-open connections in the TCB. When SYN cookies are enabled, the server encodes a new TCP connection's parameters as a cookie in the packet's initial sequence number, and refrains from allocating state for a new connection until the cookie is again received from the client and validated.

The SYN cache aims to contain TCP SYN attacks by reducing the amount of state maintained on the server for half-open connections. Although efficient against a single attacker (or a small botnet), SYN caches do no provide protection against larger botnets for which the attack rate can easily exceed the space allocated for the cache. Once the cache is full, the server will default to the same behavior it performs when its backlog limit is reached, defeating the purpose of the cache. Although SYN cookies eliminate the key target of the SYN flood attack (the TCP backlog), they do not provide protection against large botnets. Attackers in control of a large number of zombie machines with valid (non-spoofed) IP addresses can, without added effort, overload the server's `listen` queue with valid TCP requests at a rate that surpasses the server's ability to accept them. Because they only tackle the problem only on the server end, SYN cookies are not a mechanism for stripping the malicious actors of their ability to conduct exhaustion attacks; further, it is not clear how SYN cookies can be generalized to serve as protection schemes for different types of state exhaustion attacks [17].

B. Client puzzles

Cryptographic client puzzles have been proposed to counter an asymmetry in today's Internet: clients can request substantial server resources at relatively little cost. Client puzzles alleviate this asymmetry by forcing clients to commit compute power as payment for requested resources.

Client puzzles have previously been proposed as a mechanism to combat junk mail [30], website metering [31], protecting the network IP and TCP channels [16], [17], [20], [32], protecting the TLS connection setup [18], [22], protecting key exchange [19], and protecting the capabilities-granting channel [8]. In addition, client puzzles are at the heart of the mining process of today's cryptocurrencies [33], [34]. Upon receiving a SYN packet, the server computes a puzzle challenge, sends it to the client, and does not commit any resources. After receiving the challenge, the client will employ its computational resources to solve the challenge and send the

solution to the server. The server will then commit resources to the client only if the solution is correct.

Despite its promise, several challenges face the adoption of client puzzles as a practical defense measure against state exhaustion attacks. First, there is a shortage of implementations that allow for the comparison and the evaluation of different types of challenge creation and verification mechanisms. In this paper, we implement clients puzzles in the Linux kernel and provide access to our implementation as a kernel patch.

Second, an important advantage of client puzzles is the ability to influence the clients by setting an appropriate puzzle difficulty. However, there are no concrete and theoretically backed recommendations for selecting the appropriate difficulty, especially when faced with a mixture of power-endowed devices. Previous approaches [16], [18], [19], [22] suggest using a fixed, victim determined, puzzle difficulty for all clients in an effort to maintain a stateless protocol and to avoid creating a new state-exhaustion attack vector. These approaches do not however provide concrete methods to select the difficulty level in a manner that reflects the victim server's load and its clients' computational prowess. In this paper, we present a game-theoretic formulation of the difficulty selection problem incorporating the server's provisioning as well as the computational profile of its clients.

Furthermore, a fixed puzzle difficulty might lead to fairness concerns as low-powered devices have to solve the same puzzles as higher-powered ones. RFC 8019 [19] suggests providing per-IP puzzles, however, it does not specify how the difficulties should be computed nor how to avoid increasing the attack surface. The work in [17] attempts to alleviate this problem by requiring clients to place bids on the server's resources by solving increasingly difficult puzzles. In addition to violating the TCP protocol by adding more round trips, this mechanism can be exploited to target clients since it moves the puzzle initiation process from the server to the client. In this work, we present a prototype *puzzle-based queuing* approach that rewards slow sending devices with higher access priority to the server's resources. It maintains negligible state and falls back to a fixed difficulty if that state is exhausted.

Laurie and Clayton [35] present an economic analysis to argue against the use of proof-of-work mechanisms to combat email spam. We agree with the authors that computational puzzles do not possess "magical" properties that make them practical in every situation. We however argue that state exhaustion attacks do not have the same nature as spam emails. First, unlike spam, state exhaustion attacks do not depend on the involvement of human users to click on malicious links. Second, DDoS attacks have a lower cost barrier as they are launched from compromised botnet machines and not from specialized attacker hardware. We believe that our theoretical and experimental results showcase the merits of proof-of-work mechanisms in tolerating SYN and connection floods. In fact, our work complements the security analysis performed in [36]–[38] with the required protocol engineering and design, allowing for an improved understanding of client puzzles.

III. The Game-Theoretic Model

In this section, we introduce our game-theoretic model for computing the puzzle difficulties that balance the clients' computational load as well as the server's provisioning. We first present our threat model and assumptions, and then turn to discussing our game-theoretic model.

A. Assumptions and threat model

In this paper, we make the following assumptions.

Assumption 1. Common state exhaustion attacks, specifically TCP connection floods as well as higher-layer attacks, require the presence of a two-way communication channel between the attacker bots and the victim server. That is evident from the nature of the state exhaustion attacks as well as their ability to circumvent scrubbing and filtering techniques by sending lower volumes of traffic [1]. The implication is that during a single-vector state exhaustion attack, the victim server is able to receive packets from, and send packets to, its legitimate users as well as the attackers' machines. In the presence of a multi-vectored attack, we assume the presence of volumetric attack mitigation techniques (such as cloud-based protection-as-a-service); client puzzles will complement those techniques to provide DDoS defenses against hybrid attacks.

Assumption 2. We assume that the attackers can control a large number of zombie machines that form botnets to coordinate large-scale attacks aimed at depleting a target server's resources. However, we assume that the attacker's army of bots comprises commodity machines (e.g., workstations, mobile phones, and IoT devices) but not clusters of servers with large computing resources. Such clusters are part of enterprise solutions and therefore employ better protective mechanisms than commodity machines, so are harder to compromise. We further assume that the attackers can capture and replay packets, but are not able to change their content; protection against integrity attacks is beyond the scope of this paper.

The above assumptions are similar to the ones made in [16] and [17]. Moreover, client puzzles do not require the end-server to differentiate between malicious and benign traffic. In fact, the low-volume nature of state exhaustion attacks and the requirement for quick and effective protective mechanisms can impede the accuracy of anomaly detection mechanisms.

B. Difficulty selection as a Stackelberg game

We formalize the problem of selecting the puzzle difficulty similar to a network pricing problem [23]–[25]. We model the problem as a Stackelberg game between the service provider and the service users. The service provider is the leader who sets the difficulty of the puzzles that the clients must solve to receive service. The users are the followers who then choose their request rates to optimize their local utility.

Our model rests on the assumption that all clients are selfish agents seeking to optimize their local utilities; we do not specifically posit a model for malicious bots. This assumption is rooted in the following observations. First, before the attack starts, the server does not have the means to distinguish between benign clients and malicious bots. Second,

TCP by default treats every connection request it receives as a benign request, and thus sends an ACK packet back without checking whether the request came from a benign user or a compromised bot. Third, positing a specific attacker model would require the estimation of attacker preferences and utilities, which the server has no means of measuring. This could create a schism between the model and its application in the real world. We therefore treat every request as if it is benign, and capture the presence of a large botnet by obtaining the asymptotic solution for our model.

Let x_i be user i's request rate, for $i \in \{1, 2, \ldots, N\}$, where N is the total number of users in the system. Consequently, $x_{-i} = \sum_{j \neq i}^{N} x_j$ is the total request rate of all the other users. Our model captures the puzzle's difficulty by using the expected number of hash operations needed to find and verify its solution. Let p_i be user i's puzzle; $\ell(p_i)$ is then the expected number of hash operations that user i has to perform to find a solution to p_i. Let $S(\bar{x} = \sum_i x_i)$ be the expected service time for a user's request. User i's utility can be written as

$$u_i\left(x_i, x_{-i}, p_i\right) = w_i \log(1 + x_i) - \ell\left(p_i\right) x_i - S(\bar{x}) \qquad (1)$$

w_i is a user-specific parameter that models the user's valuation of the provider's service. In other words, w_i represents the amount of work user i is willing to pay per request. $\log\left(1 + x_i\right)$ represents the user's expected benefit when making decisions under risk or uncertainty [23], [39]. The utility function can be interpreted as the difference between the user's expected benefit and the amount of work she has to expend to solve a puzzle per request added to the expected service delay she incurs. Each user, being a rational and selfish agent, will choose a request rate that optimizes her local utility. That will lead to the users adopting the Nash Equilibrium (or simply, equilibrium) rates x_i^* for $i \in \{1, 2, \ldots, N\}$ such that

$$u_i\left(x_i^*, x_{-i}^*, p_i\right) \geq u_i\left(x_i, x_{-i}^*, p_i\right), \ \forall x_i > 0, \forall i \qquad (2)$$

The service provider's problem is to find a puzzle difficulty such that (1) it can effectively reduce the impact of state exhaustion attacks and (2) minimize the amount of work the server does to generate and verify puzzles. Let $\mathcal{P}$ be the space of all possible cryptographic puzzles and $g(p_i)$ and $d(p_i)$ be the expected numbers of hash operations that the provider needs to perform to generate and verify a solution to puzzle p_i, respectively. We model the provider's problem as finding the puzzles $\mathbf{p}^* = \{p_i^* \in \mathcal{P},\ i \in \{1, 2, \ldots, N\}\}$ such that

$$\mathbf{p}^* = \underset{\mathbf{p} \in \mathcal{P}^N}{\arg\max} \sum_{i=1}^{N} \left(\ell(p_i) - \left(g(p_i) + d(p_i)\right)\right) x_i^* \qquad (3)$$

Equation (3) captures the provider's goal of maximizing the amount of work that the clients have to perform to obtain service under attack while minimizing the amount of work it must perform to generate puzzles and verify solutions. This formulation, in fact, captures the trade-off between the puzzle's complexity and the expected work that the provider needs to perform to generate and verify puzzles. The tuple $(\mathbf{x}^* :=< x_1^*, x_2^*, \ldots, x_N^* >, \mathbf{p}^* :=< p_1^*, p_2^*, \ldots, p_N^* >)$ represents the solution to the full Stackelberg game.

IV. Application to the Juels puzzle scheme

We now show how the framework we introduced in Section III can be applied to the puzzles protocol presented in [16]. We first describe the puzzles protocol from [16] and then show the solutions we obtain using our framework. For our modeling and analysis, we assume that the server issues puzzles with the same difficulty for all of its clients, i.e., $\ell(p_i) = \ell(p_j)\ \forall i, j \in \{1, 2, \ldots, N\}$. This assumption ensures a stateless protocol, follows the IETF TLS puzzles draft [18], and is recommended in previous work [16].

A puzzle in this scheme is a bitstring of length l bits having $m < l$ bits of difficulty. The puzzle-issuing server starts by creating the hash $y = h\left(s, T, \text{packet-level data}\right)$, where s is a secret key; T is a timestamp; the packet-level data are a concatenation of the source and destination IP addresses and ports; and h is a collision-resistant hash function. The server challenges a client to provide k solutions to a puzzle P formed by the first l bits of y.

Upon receiving P, the client computes, by brute force, k solutions $\{s_1, \ldots, s_k\}$ such that for $1 \leq i \leq k$, $|s_i| = l$ and the first m bits of $h(P, i, s_i)$ match the first m bits of P, where h is the same hash function that the server used. The client then sends the solutions back to the server, which in turn, verifies their validity and subsequently accepts the request.

The solution: Since obtaining a single solution of length m bits is best done by brute force, it requires a maximum of 2^m and an average of 2^{m-1} hashing operations. Since each puzzle requires k solutions, solving a puzzle then requires an average of $k \times 2^{m-1}$ hashing operations. Therefore, for each user $i \in \{1, 2, \ldots, N\}$, $\ell(p_i) = k \times 2^{m-1}$.

To capture the expected service time for the users, called $S(\bar{x})$, we abstract the server's operation with an $M/M/1$ queue with a service rate μ. We argue that this abstraction is enough for our purpose, since the attacks in which we are interested target the TCP stack and are independent of the application that the server is running; they are affected only by the application's ability to remove established connections from the accept queue. The service rate μ can be obtained by running stress tests on the application provider's infrastructure and can capture different service optimizations such as replications and caching. Subsequently, we express the expected service time as $S(\bar{x}) = \frac{1}{\mu - \bar{x}}$, when $\bar{x} < \mu$. This condition assumes that the server is well-provisioned to handle the users' load under regular conditions. We therefore rewrite Equation (1) as

$$u_i\left(x_i, x_{-i}, p_i\right) = w_i \log(1 + x_i) - k \times 2^{m-1} x_i - \frac{1}{\mu - \bar{x}} \qquad (4)$$

We now turn to the provider's formulation. We represent the space of all possible puzzles as the set of tuples (k, m) where $k \in \mathbb{N}$ is the number of solutions requested and $m \in \mathbb{N}$ is the number of bits of difficulty in each. Therefore we write $\mathcal{P} = \{(k, m), k, m \in \mathbb{N}\}$. As previously discussed, every challenge can be generated using only one hash operation; therefore, we write $g(p_i) = 1,\ \forall i$.

When the server receives a solution, it generates a hash from the received packet's header and then verifies each of the k solutions until it finds a violating one or deems the puzzle correctly solved. If the server chooses which of the k solutions to verify uniformly at random, it then needs an average of $\frac{k}{2}$ hashing operations. Therefore, we can write $d(p_i) = 1+\frac{k}{2}$, $\forall i$.

Since we assume that the service provider issues puzzles with the same difficulty for all users, we henceforth write $p = (k, m) = p_i, \forall i$. We can then rewrite Equation (3) as

$$p^* = \arg\max_{p \in \mathcal{P}} \sum_{i=1}^{N} \left(k \times 2^{m-1} - 2 - \frac{k}{2} \right) x_i^*(p) \qquad (5)$$

Let w_{av} be the average client valuation of the server's service and α be the server's asymptotic service rate per user, under normal operation.

Theorem 1. *The Nash equilibrium is achieved at* $p^* = (k^*, m^*)$ *such that:*

$$\ell(p^*) = k^* \times 2^{m^*-1} = \frac{w_{av}}{(\alpha + 1)} \qquad (6)$$

Proof. For brevity, we include the full proof of our theorem in the online version of the paper at `https://github.com/nouredd2/puzzles-utils`. □

Analysis: The equilibrium difficulty we obtained in Theorem 1 illustrates an important design tradeoff between the server's provisioning and the difficulty of the puzzles that the clients should solve when the server is under attack. A well-provisioned server, i.e., one for which $\alpha > 1$, will be able to absorb a larger fraction of the attack and subsequently ask its clients to solve less complex challenges. In that case, the clients help the server tolerate the attack and commit fewer resources than they are willing to — the average number of hashes they would need to perform to solve a challenge is less than w_{av} — so the client achieves high utility. On the contrary, a server that is not able to handle all of its clients' regular load, i.e., one for which $\alpha < 1$, would require its clients to solve harder puzzles ($p^* \simeq w_{av}$) and thus achieve lower utility levels. Therefore, to tolerate an attack, the server asks its clients to commit more resources, risking the dropout of more clients as the intensity of the attack increases. Those clients with $w_i < w_{av}$ would consider it best for them to drop out, since it would be too costly as a function of the resources committed to obtain a connection.

We further note that our model and solution are agnostic to the application that is run by the server. That, in fact, is consistent with TCP being a transport-layer protocol that is independent of the type of application running on top of it. All our model requires is an estimate of the server's capacity to handle large loads (i.e., the parameter α), which can be obtained by running appropriate stress tests. Server replication and load balancing are then captured in our model through an increase in the value of α (given the same load).

Finally, we note that our result is not affected by the presence of long-lived TCP connections (for example, if HTTP/1.1 [40] is being used). The puzzles protect the TCP connection establishment channel and allow users to connect to the server in the presence of malicious attacks. The lifetime of the established connection is not affected by the presence or absence of puzzles; in the case of HTTP/1.1, the goal of the challenges is to allow clients to establish the TCP connection upon which the HTTP session persists.

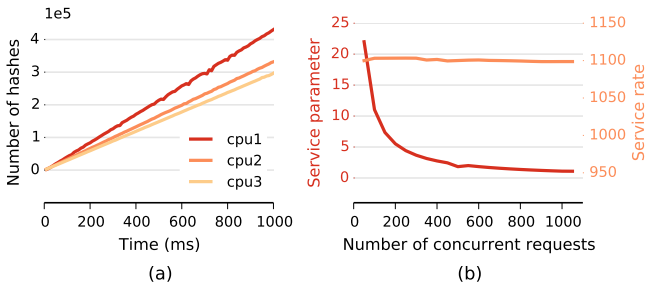


Fig. 1: Profiles of (a) client (w_{av}) and (b) server (α).

Obtaining model parameters: The model parameters, w_{av} and α, relate to the performance capabilities of the server and the clients. We first describe an experimental procedure for obtaining the model parameters. Then we discuss how we applied the procedure to an experimental setup to demonstrate the Nash strategy.

First, w_{av} is the number of hashes we assume the client is willing to perform to complete the TCP handshake. It represents the level of acceptable service degradation as each TCP connection will take longer to finish. To find w_{av}, we assume that 400 ms is adequate time to establish a TCP three-way handshake for a legitimate client when the server is under attack. Usability studies show that a 400 ms delay does not interrupt the user's flow of thoughts [41]. Using that assumption, we find the number of hashes a machine can perform in 400 ms by profiling the machines. w_{av} is the average value obtained during the experiments.

Second, α is the service parameter of the server. It is directly related to the processing power of the server. To obtain the parameter, we start by stress testing a server. The stress test varies the rate of requests per second and records the time it takes to get service for each rate. We compute α as the ratio of the service rate over the number of concurrent requests.

Finally, after obtaining w_{av} and α, we calculate the equilibrium difficulty parameters (k^*, m^*) by using Equation (6). The choice of those parameters exposes a trade-off between the number of hashes the server needs to verify a solution and the probability that an adversary can guess a solution. Choosing a very small k will increase the attacker's ability to guess a solution, and selecting a large k will increase the solution verification time. On the other hand, if lower values of k are selected, the challenge difficulty m would increase allowing the server to offset its lack of computational resources by asking its clients to solve harder challenges.

Example: In the following, we present an example of computing the Nash equilibrium difficulty for a server serving a variety of machines with varying processing powers. Starting with the client, we obtain w_{av} by profiling the number of

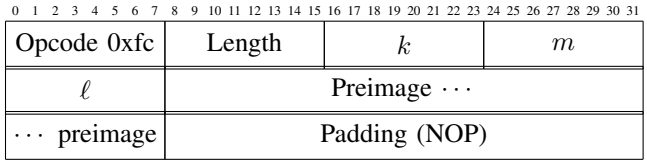


Fig. 2: TCP Options block for a SYN challenge.

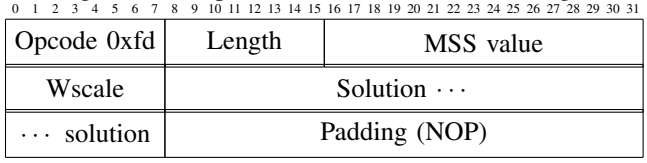


Fig. 3: TCP Options block for a SYN solution with $k = 1$.

SHA-256 operations per second. Figure 1(a) shows the profile of three CPU types: (1) `cpu1` is an Intel Xeon E3-1260L quad-core processor running at 2.4 GHz, (2) `cpu2` is an Intel Xeon X3210 quad-core processor running at 2.13 GHz, and (3) `cpu3` is an Intel Xeon processor running at 3GHz.

The average number of hashes that can be performed over the three types of CPUs is $w_{av} = 140630$. Although the CPUs we profiled are not an exhaustive representative set of the processing powers of a typical clientele, in all of our experiments, we leveled the playing field by providing all the attackers with similar or better computational powers.

Then we estimate the server's α parameter. We deployed an Apache2 web server on a dual Intel Xeon hexa-core processor running at 2.2 GHz with 24 GB of RAM. We then used the Apache benchmarking tool `ab` [42] to profile the performance of the server under regular and high loads. Figure 1(b) shows the service rate μ and the service parameter α of our server as the number of concurrent requests attempted by `ab` increases. Our server was able to maintain a constant service rate under high load ($\mu \simeq 1100$ requests/s), and thus the parameter α converged to a value of 1.1 as the load increased. Thus for our example, with $w_{av} = 140630$ and $\alpha = 1.1$, the TCP puzzle difficulty is set at $(k^* = 2, m^* = 17)$.

V. Implementation

We implemented the TCP challenges in the TCP stack of the Linux 4.13.0 kernel. The puzzles are turned off by default and are only enabled when the socket's queue is full. We designed our implementation in such a way that the challenges take precedence over the SYN cookies once the queue is full; we do, however, support SYN cookies as a backup option. We provided support for dynamic tuning of the parameters of the challenges (k, m) through the kernel's `sysctl` interface.

We generate the challenge's pre-image by hashing a string containing (1) a server's secret key, generated once at the start of a socket's lifetime, (2) the server's current timestamp, (3) the SYN packet's source and destination IP addresses, and (4) the packet's source and destination port numbers. We used the Linux kernel's SHA256 hashing function, since it provides the necessary pre-image resistance guarantees [16].

To avoid breaking the TCP definition, we inject the challenges and solution into the options field of the TCP SYN-ACK and ACK packets. Figure 2 shows the format of the TCP option we implemented to transmit a challenge in the SYN-ACK packet. We chose an unused opcode (0xfc) to represent a challenge option. The `Length` field indicates the length of each option block in bytes, including the opcode and the field itself. We allocate one byte each for the number of solutions k, the difficulty of the puzzle m (in bits), and the pre-image and solution length l. Next, we insert the challenge's pre-image. Finally, following the TCP stack requirement, each option block must be 32 bits aligned, we, therefore, insert 0 to 3 NOP fields to ensure alignment.

Figure 3 shows the format of the TCP option used by a client to send a solution. Much as in the challenge option, we made use of unallocated opcode (0xfd). Since the server keeps no state about the client after receiving the first SYN packet, the client safely assumes that the server has ignored its previously announced *Maximum Segment Size* (MSS) and *Window Scaling* (Wscale) parameters. We then resend the MSS and Wscale values within the solution, write down each of the k solutions, and perform alignment to 32 bits.

The benefits of adding the MSS and Wscale parameters to the solution option block are twofold. First, it means that the challenge protocol will be self-contained; implementation of the TCP stack usually ignores all options other than timestamps in any packet other than the SYN and SYN-ACK packets. Therefore, support for the challenge protocol does not require changes to legacy options parsing. The addition also provides us with the benefit of reducing the space needed to resend the options in the ACK packet. For example, sending the MSS values as a separate option would require 4 bytes, while we need only 2 in the case of the self-contained solution option. Second, we encode the MSS value by using 16 bits (as defined in the specification of TCP), instead of the 3 bits provided by SYN cookies. In addition, when SYN cookies are in place, the client and the server cannot agree on the window parameters, which reduces the performance of the TCP connection.

Also, modern TCP implementations support the exchange of timestamps as options in the TCP header. Our implementation makes use of the timestamps, whenever available, to generate, solve, and verify challenges. However, if the timestamp option is not enabled (for example, it was disabled by the client or the server), our implementation embeds the timestamp used in the generation of the challenge (an additional 4 bytes) in both the challenge and the solution packets.

Furthermore, when the server's accept queue overflows, its default behavior is to reject new connections, even if the protection mechanism is in place. However, for our purposes, since the goal of the puzzle protection mechanism is to throttle the rate of all clients (both benign and malicious), we modified the listening TCP socket's implementation to send a challenge when the protection is in effect, even if the accept queue overflows. When the server receives an ACK packet while under attack, it first checks whether the queue is full and performs the verification procedure only if there is room to accept the connection. If the queue is full, the server will ignore the ACK packet. In such a case, the user

(whether benign or malicious) assumes that the connection has been established and will begin sending application-level packets thus causing the server to reply with a reset (RST) packet to signal that the connection was not established. This implementation choice achieves the goal of deceiving the malicious users into thinking that they have established a connection when they have not; the malicious agents that do not send application-level packets will not receive a RST packet to indicate that the server has dropped the connection.

Finally, to combat replay attacks, we make use of the timestamp in the solution to check whether a challenge has expired. This stateless mechanism hinders an attacker's ability to replay solution packets, since tampering with the timestamp will cause the solution verification to fail. The timeout interval can be tuned through the kernel's `sysctl` interface.

VI. Evaluation

Using our modified Linux kernel, we evaluated the performance of the TCP puzzles in safeguarding a server TCP connection establishment channel from state exhaustion attacks.

We performed the experiments using the DETER [43] cybersecurity testbed. In the spirit of moving towards a "science of security" through reproducible experiments [44], we provide all of our experiment scripts and datasets online at `https://github.com/nouredd2/puzzles-utils`.

The goal of our experiments is to evaluate (1) the effectiveness of TCP puzzles in protecting against state exhaustion attacks, (2) the impact of TCP puzzles on service quality, and (3) the ability of the Nash equilibrium puzzle difficulty to balance the client solution and the server verification load as well as its ability to effectively rate-limit attackers. Our victim server runs an HTTP application that accepts "`gettext/size`" requests and returns messages that contain `size` random bytes. The clients run an HTTP client that requests text from the server at a prespecified rate.

In a real-world deployment, service would be provided by a farm of servers, but our scenario uses only one server and a smaller set of clients. In larger systems, since a load balancer forwards TCP connection requests to individual servers, an attack has to ensure that its wave of requests reaches all of the servers to effectively deny service. Therefore, adding more servers allows a service provider to tolerate bigger attacks by large botnets. Our results show that a server using TCP puzzles as a means for state exhaustion DDoS protection can tolerate a larger botnet than an unprotected server can. We hence argue that when all the servers in a farm employ our protection, the system will be able to tolerate a larger botnet that is proportional to the improved tolerance of a single server. Our experiment scenario thus studies the protection offered to a single server; the results are to scale when more load-balanced puzzles-equipped servers are deployed.

We consider two types of attackers. The first uses randomized source IP addresses to target the server's `listen` queue with a flood of half-open TCP connections (using `hping3`). The second type uses real IP addresses to flood the server with established connections (using `nping`) in an attempt to

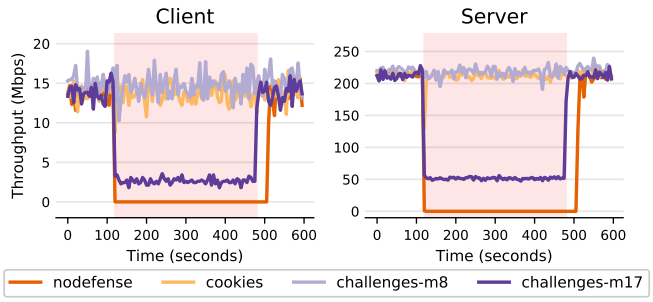


Fig. 4: Throughput at a client and server during SYN flood.

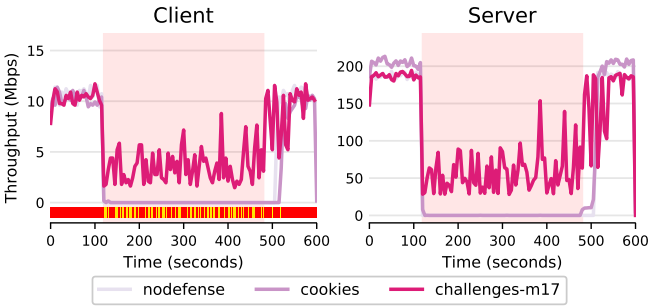


Fig. 5: Client and server throughput during a connection flood.

fill its `accept` queue and prevent new, legitimate connections from being established. Unless otherwise stated, we use the following experiment parameters. The set of clients contains 15 machines requesting $10,000$ bytes of data at exponentially distributed time intervals, with rate $r_c = 20$ requests per second. The botnet consists of 10 machines running an attack at a constant rate $r_a = 500$ requests per second, amounting to an overall attack rate of $5,000$ packets per second (pps). All of the malicious machines are equipped with a computational power equal to, or greater than, that of the clients' machines. Except in Experiment 4, all of the machines in our setup were equipped with our modified kernel.

Finally, except for Experiment 5, all the experiments used the same network topology with well-provisioned link bandwidths so as to avoid link saturation. The backbone consisted of three routers fully connected with 1 Gbps links. The server connected to the network with a 1 Gbps link, while all the other hosts connected to the network with 100 Mbps links. All of our agents ran on physical machines with Ubuntu 16.04 LTS along with our patched Linux 4.13.0 kernel. We deployed the packet-monitoring software, `tcpdump`, on all of the machines, and used the captures to measure the throughput at the server, the throughput at each host, and the number of dropped TCP connections. We report here on the throughput since it represents a direct assessment of the impact of puzzles on our application; nevertheless, we acknowledge that different applications will require different metrics.

A. Experiment 1: SYN and connection flood protection

In the first scenario, we started a distributed SYN flood attack. Without protection, the SYN flood filled the `listen` queue with half-open TCP connections, leading the server to

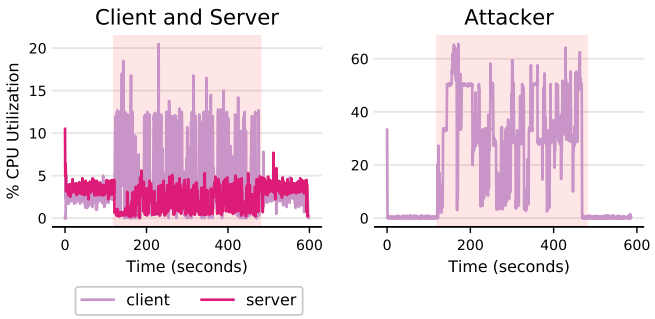


Fig. 6: CPU utilization during a connection flood attack.

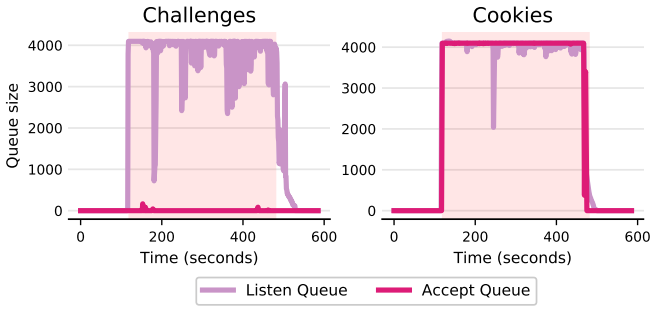


Fig. 7: Queue sizes during a connection flood attack.

drop new incoming connections. We measured the throughput at a client and the server for three settings: (1) no protection (control settings), (2) TCP SYN cookies, and (3) TCP client puzzles. Figure 4 shows the throughput measured during the experiment. The attack duration, shown by the shaded region, was initiated at $t = 120$ and concluded at $t = 480$. The throughput's behavior for both the server and client was consistent; we therefore restrict our analysis to the server's case. For the control setting, the server's throughput dropped to zero as soon as the attack started and returned to full capacity 30 seconds after the attack ended. On the other hand, SYN cookies were effective at rendering the attack ineffective and ensuring a constant throughput at the server throughout the attack. By storing partial state of the connection in the TCP sequence number instead of in the `listen` queue, SYN cookies provide protection against this type of attack. Finally, when low difficulty puzzles are enabled, $(k, m) = (1, 8)$, the throughput is unaffected during the attack. Similarly to SYN cookies, the puzzles enabled reconstruction of a connection's state with no use of the `listen` queue. However, when we used the Nash equilibrium difficulty, $(k, m) = (2, 17)$, the throughput was reduced to 50 Mbps during the attack. This reduction occurred because the equilibrium strategy is more aggressive than the easier setting; in this scenario, easy puzzles were enough to alleviate the attack as the botnet was not completing the connection.

For the second scenario, we used the attacker nodes to launch a distributed connection flood attack. We measured the same metrics as in the first scenario for three cases: no protection, SYN cookies, and TCP puzzles at Nash difficulties. The TCP puzzles at a difficulty of 8 bits were ineffective at

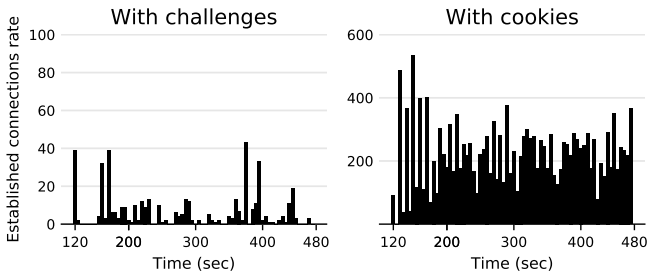


Fig. 8: Effective attack rate during a connection flood attack.

protecting the server's state. For readability, we elected not to show these results in this plot since we will revisit various difficulty settings in Section VI-B.

Figure 5 shows the throughput of a client and the server during the experiment. We used the sparkline in the client plot to mark when the server sent a SYN-ACK packet with a challenge (bright tick) or without a challenge (dark tick). The results show that SYN cookies are ineffective during a connection flood; the server's throughput drops to 0, as it would if no protection were in place. In both those cases, the server needs 30 seconds to detect the end of the flood and fully recover. On the other hand, TCP puzzles at Nash difficulties provide tolerance of the flood attack. The throughputs of the client and the server were about 40% of their respective nominal rates. It is interesting to note that the throughput periodically spiked during the attack phase. This occurs because not all the requests of the clients required a puzzle, as shown by the dark ticks in the sparkline during the attack phase. The performance improvement was due to the opportunistic nature of the protection controller; that is, when the `listen` queue was not full, connection requests were answered without a challenge, allowing a host to take advantage of the resource instantly. We also note that easy puzzles were unable to affect the attacker bots' connectivity rates and thus provided no better protection than SYN cookies did.

In addition, we measured the impact of the TCP challenges on the CPU utilization of the client, server, and attacker machines. Figure 6 shows that the impact on the server of generating and verifying the puzzles was negligible; the server's CPU utilization stayed below 5% and did not exceed its nominal (under regular load) value. In accordance with the nature of computational puzzles, the CPU utilization at the clients' machines increased during the attack, but still remained well under 20%, with an average of 10%. The attacker machines, on the other hand, witnessed a spike in CPU utilization during the period of the attack, reaching a maximum of 60%. These results show that our equilibrium difficulty setting achieved our desired goals of (1) putting minimal overhead on the server in generating and verifying puzzles, (2) inducing tolerable nuisance for the clients, and (3) effectively rate-limiting the attackers' established request rate and increasing their computational burden. In fact, the sudden increase in the CPU utilization at the botnet machines can alert the owners of these machines to the presence of malware.

We further studied the impact of the TCP cookies and puzzles on the server's `listen` and `accept` queues during a connection flood attack. Figure 7 shows that when SYN cookies were the only defensive mechanism in place, both queues were fully saturated, which explains the zero throughput observed by the benign clients. On the other hand, with TCP challenges in place, the `accept` queue was almost always empty, which was a direct result of the puzzles' ability to rate-limit every user, whether benign or malicious, to an average of 2 requests per second. In addition, the `listen` queue, although mostly saturated, showed frequent openings that are consistent with the opportunistic nature of our implementation, as indicated by the sparklines in Figure 4.

Finally, we showed that TCP puzzles (at Nash difficulty) throttled the attacker's rate of established connections. We measured the effective completed connection rate of all attackers as seen by the server during the connection flood. The measurements, shown in Figure 8, reveal that the attack rate was not affected by TCP cookies, achieving an average rate of 225 connections per second (cps), whereas puzzles severely limited the attackers' rate down to an average of 4 cps, a reduction by a factor of 37.

B. Experiment 2: Nash equilibrium strategy

In this experiment, we showed that the Nash equilibrium difficulty provides the optimal balance between the clients' throughput and the attack tolerance during an attack. We selected the Nash equilibrium based on the capabilities of the clients and the server's defense requirements.

Figure 9 shows the average and standard deviation of the throughput of a client during an attack. In general, for any k, if $m < 12$, the ease of solving the challenges did not affect the attackers' rate, so a denial of service occurred. The Nash equilibrium strategy resulted in the most stable throughput, with an average of 3.90 Mbps and low variability. Even though some of the other settings had a higher average throughput, their throughput was highly unstable, reaching zero at many times. Further, we note that when the difficulty was set to $(k = 2, m = 16)$, the throughput achieved a slightly better average with comparable variability. However, the Nash difficulty setting provided the rate that balanced the acceptable cost a client was willing to pay and the server's ability to tolerate state exhaustion attacks by throttling the attackers' rates. In fact, at the Nash difficulty, the puzzles mechanism reduced the attackers' average SYN sending rate from 2250 pps for $(k = 2, m = 16)$ to 1668 pps, and the average connection establishment rate from 30 cps to 22 cps.

C. Experiment 3: Botnet effectiveness

In the third experiment, we showed that TCP puzzles increased the server's tolerance to a botnet and required attackers to increase their botnet's size to deny service. We varied the botnet's size and attack rate and measured the cumulative attack rate as seen by the server, referred to as the *connection completion rate*. The connection completion rate is the effective attack rate that actually impacts the server. In the first scenario, we set the number of nodes in the botnet to 5 and varied the sending rate of each node between 100 and 1000 pps. Figure 10a shows the rate of completed connections as the rate of each attacker machine was varied. The results show that the TCP puzzles were capable of rate-limiting the effective attack rate. As the per-node attack rate increased, the effective attack rate was limited to 11 cps in all cases.

In the second scenario, we varied the number of machines in the botnet while setting the cumulative attack rate to $5,000$ pps; each machine's rate was set at $5,000/(\text{size of botnet})$. Figure 10b shows the measured effective attack rate as the number of machines was varied. The results show that attackers had to increase the size of their botnets to increase their effective attack rates. The effective attack rate, although it linearly increased with the increase in the number of attack machines, only peaked at 25 cps. In contrast to the near-constant rate in the first scenario, the effective attack rate increased in this scenario since more machines were enlisted in the botnet. However, this increase did not reflect the increase in resources being committed to the botnet. At this rate of increase, a botnet has to commit 500 machines to reach an effective attack rate of 5000 cps.

In conclusion, the attacker cannot increase the effective attack rate by increasing the individual rates; she has to increase the number of machines in the botnet. TCP puzzles at the Nash equilibrium difficulty significantly increased the cost of a state exhaustion attack.

D. Experiment 4: Adoption of TCP puzzles

In this experiment, we showed that a client solving the TCP puzzles is almost always able to connect to the server regardless of whether the attacker elects to solve or ignore the puzzles or select a combination thereof. On the other hand, a client that does not solve puzzles gets erratic service when the attacker is solving the puzzles and almost no service when the attacker floods the server without solving any puzzles. In this experiment, we used machines that were not patched to support TCP puzzles; we tested four scenarios in which (1) neither the attacker and the clients solved puzzles (NA,NC); (2) the attacker solved puzzles while the clients did not (SA,NC); (3) both the clients and the attacker solved puzzles; and (4) the clients solved puzzles and the attacker did not. We group scenarios (3) and (4) together and label them (*A, SC). Figure 11 shows the percentage of completed connections for all the proposed scenarios. We observe that a client solving puzzles is not denied service regardless of the attacker's type; this happens because the attacker, being rate-limited when solving puzzles and having its requests ignored when not solving, is not able to fill the `accept` queue of the server. On the other hand, a non-solving client faced with a solving attacker experiences a highly variable percentage of completed connections, reaching 0 at some instances. The reason is the opportunistic nature of the puzzles controller (as observed in Experiment 2); the rate-limiting impact on the attacker machines can empty slots in the server's queues, thus providing openings in which the non-solving client can establish

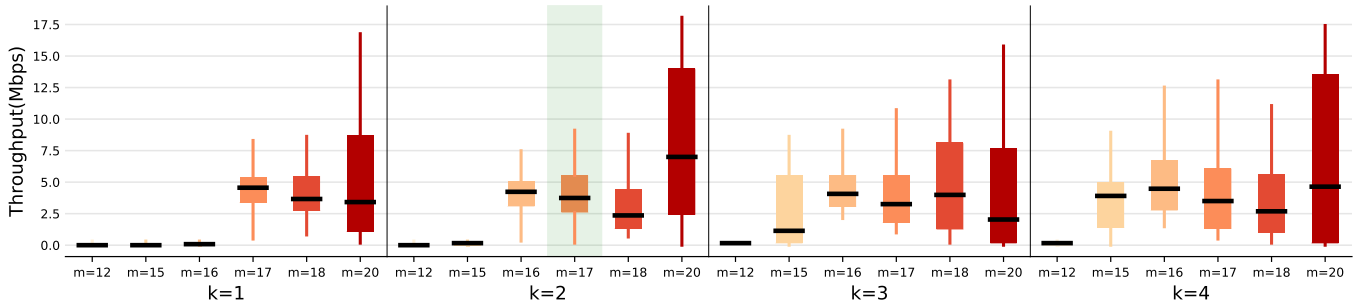
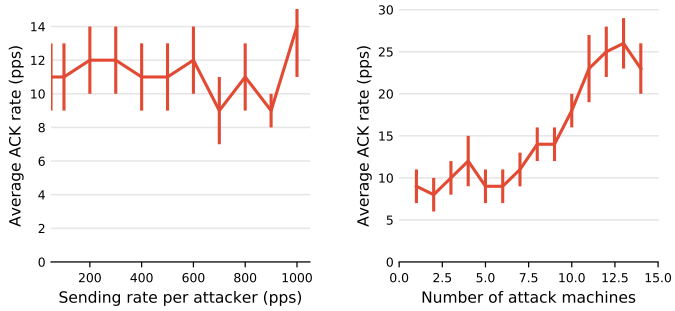


Fig. 9: Box plot of the client throughput for different puzzle difficulties.



(a) Impact on flooding rate (b) Impact on botnet size

Fig. 10: Impact of the puzzles on the attack as the size of the botnet and the flooding rate are varied

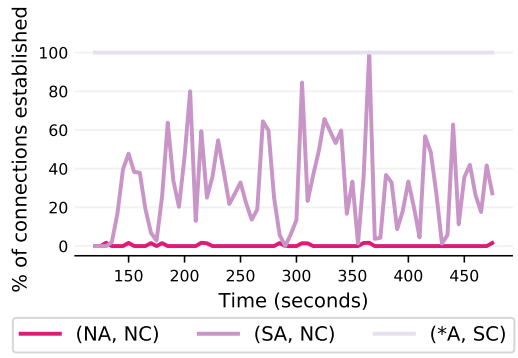


Fig. 11: Percentage of established connections when TCP puzzles adoption is not complete

connections. However, when faced with an attacker that does not solve the challenges, the non-solving clients are denied service, because the attacker's vast resources beat the clients' requests for the resources freed by the puzzles controller. We note that the service promises provided by our implementation to noncompliant clients are similar, and sometimes better than, those provided by network capabilities [10].

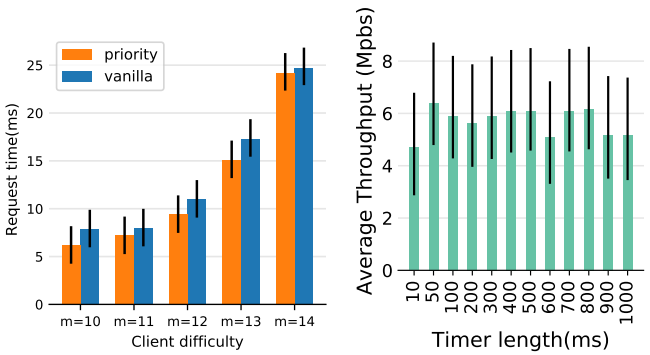
E. Experiment 5: Puzzle-based Queueing

In this experiment, we present a prototype solution that addresses the challenge that arises from treating every user as potentially malicious, namely the possibility that more powerful attackers will be able to control the majority of the server's resources. To that end, we modified the queueing behavior of the kernel's TCP stack from a *First In First Out* (FIFO) discipline to a *priority-based* behavior. Our design rests on the observation that a benign user would have an average of $\hat{n}_i = \frac{2^{m_i-1}}{\beta_i}$ seconds between their requests, where β_i is that user's hashing rate per second. So a user that can send more than one request in a single $\hat{n}_i$ interval is likely to be attempting to launch a connection flood attack.

We therefore allow each user i to solve puzzles at any difficulty $m_i > m^*$, and then tag that user's request with a weight $\eta_i = \lfloor \frac{m_i}{x_i} \rfloor$, where x_i is the user's request rate in a given time interval ΔT. Computing the weights for each request requires the server to keep state about incoming requests, which might also be a target of a state exhaustion attack. However, our design safeguards against this vulnerability in the following manner. First, for each user i, we need only $\lceil \log_2(m_{\max}) \rceil$ bits to keep track of x_i, where $m_{\max}$ is the maximum possible puzzle difficulty bits (32 bits in our case). This is valid since when $x_i > m_{\max}$, the weight is always 0, and thus we do not need to keep incrementing x_i. We thus achieve a significant reduction in state compared to that for a half-open TCP request. Second, we associate each entry in our state with a timer that expires after ΔT microseconds and deletes that entry. If we receive another request from the same user before the timer expires, we reset the timer for another ΔT microseconds. Thus, our state will be used for only potentially malicious users that are sending requests at a rate greater than $\frac{1}{\Delta T}$. The server can control the timer interval ΔT through the kernel's `sysctl` interface. Finally, if the server's state table is full, it assigns the same weight to all requests and falls back to the same case as in Experiment 2.

We first evaluated the performance impact of performing priority queueing for every incoming request. We compared the average service time per request for 5 clients that were simultaneously connecting to the server, each solving at a difficulty of 10, 11, 12, 13, and 14 bits, respectively. We differentiate between two cases: (1) the *vanilla* case in which we used the FIFO queue, and (2) the priority queueing implementation. Figure 12a shows the average service time for a single request in each case and indicates that, with 95% confidence, our implementation incurs no performance penalty.

We then evaluated the effectiveness of our design when it faced a mixture of users solving at different difficulties.



(a) Performance evaluation (b) Throughput at a benign client

Fig. 12: Evaluation of our prototype puzzle-based queueing

We fixed the server's minimum acceptable difficulty at $(k = 2, m = 15)$ and used 8 attacker machines and only 2 client machines (thus giving the majority of the hashing power to the attackers). The client machines solved at the minimum allowable difficulty, while the attackers used difficulties ranging from 15 to 19 bits. We varied the state table timer (ΔT) from 10 ms to 1000 ms and measured the throughput at the client.

Figure 12b shows the throughput at a benign client as the timer interval ΔT was increased. We notice that in all cases, the client was able to connect to the server and reach a throughput of at least 4.5 Mbps. That is a significant improvement over the scenario presented in Experiment 2, in which for $(k = 2, m = 15)$ the clients were not able to connect to the server (Figure 9). We also note that the length of ΔT had little impact on the observed client throughput. However, when $\Delta T = 10$ ms, the client saw its lowest throughput. The reason was that the client was attempting to connect to the server at a rate of 20 packets per second (i.e., a new connection every 50 ms). Therefore, for $\Delta T < 50$ ms, the client's timer was always renewed, and thus the weight assigned to it quickly reached 0 and its requests were treated as malicious.

VII. Limitations and Discussion

In this section, we discuss the challenges facing the adoption of client puzzles and provide an analysis of their limitations.

Software adoption: As showcased by our experiments, there is a great benefit for servers to adopt client puzzles as a mechanism for tolerating state exhaustion attacks. By rate-limiting users and protecting the server's queues, client puzzles present service providers with a chance to provide continuous service during state exhaustion attacks. Our implementation has several features that make it easy to adopt. First, a server can easily support client puzzles by simply patching its kernel. Second, our patch does not introduce any changes to the normal operation of the server and sends challenges only when the queues overflow; the TCP stack remains intact otherwise. Finally, our patch is compatible with earlier versions of the Linux kernel provided they support cryptographic operations.

While servers are incentivized to adopt TCP puzzles by the increased tolerance of attacks, clients, on the other hand, benefit from the promise of receiving service even during attacks. As shown in Experiment 4, users that enable support for client puzzles are always able to connect to the server. The users that choose not to adopt the challenges still receive full service under regular load. However, during an attack, those users will be in contention with the nonsolving attackers for the spots are freed up by the server's opportunistic challenges controller. That scenario is no worse than the case when no challenges are applied. Note that our theoretical formulation validates this observation; a user that does not adopt challenges is similar to one that values the server's services at $w = 0$.

Replay attacks: Since the server does not retain state about an incoming connection before receiving a valid challenge solution, an attacker might capture legitimate clients' solutions and replay them to overflow the server's `accept` queue. We note, however, that for a replayed solution to be validated, the attacker must retain the packet's parameters (IP addresses, port numbers, and timestamps). Therefore, a replayed solution can only be used to occupy one slot in the server's queue at a time. In addition, our implementation ensures that puzzles expire after a set timeout interval. The timeout interval limits an attacker's ability to carry out a replay flood effectively, and thus our implementation is resistant to such attacks.

Fairness and power considerations: Finally, client puzzles research faces an important challenge arising from the presence of a nonuniform mix of power-limited (e.g., mobile phones, IoT devices) and power-endowed (e.g., GPU-enabled desktops) benign devices. Motivated by the work in [15], in Experiment 5, we built a prototype design that is intended to combine puzzle difficulty and request rates to assign weights to clients' requests. Our preliminary results are promising, and we plan to explore this further in the future.

VIII. Conclusion

In this paper, we presented a theoretical formulation and implementation of client puzzles as a means for tolerating state exhaustion attacks. We addressed the challenge of selecting puzzle difficulties by modeling the problem as a Stackelberg game in which the server is the leader and the clients are the followers. We obtained the equilibrium solution that illustrates a tradeoff between the clients' valuation of the requested services and the server's service capacity. We then provided our implementation of the puzzles as a Linux kernel patch and evaluated its performance on the DETER testbed. Our results show that client puzzles are an effective mechanism that can be added to our arsenal of defenses against DDoS attacks.

Acknowledgments. We are grateful for the thoughtful comments and suggestions offered by our shepherd, Doug Blough, and the anonymous reviewers. We would like to thank Jenny Applequist for her editorial comments. This material is based upon work supported in part by the Department of Energy under Award Number DE-OE0000780, in part by the Office of Naval Research (ONR) MURI Grant N00014-16-1-2710, and in part by US Army Research Laboratory (ARL) Cooperative Agreement W911NF-17-2-0196. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

REFERENCES

[1] P. Alcoy, S. Bjarnson, P. Bowen, C. Chui, K. Kasavchenko, and G. Sockrider. (2017) Insight Into The Global Threat Landsace: NetScout Arbor's 13th Annual Worldwide Infrastructure Security Report. [Online]. Available: https://pages.arbornetworks.com/rs/082-KNA-087/images/13th_Worldwide_Infrastructure_Security_Report.pdf

[2] S. Hilton. (2016, October) Dyn analysis summary of Friday October 21 attack. Oracle Dyn. [Online]. Available: http://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/

[3] Global DDoS threat landscape: Q4 2017. Imperva Incapsula. [Online]. Available: https://www.incapsula.com/ddos-report/ddos-report-q4-2017.html

[4] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the Mirai botnet," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[5] B. Krebs. (2016) Source Code for IoT Botnet 'Mirai' Released. [Online]. Available: https://krebsonsecurity.com/2016/10/source-code-for-iot-botnet-mirai-released/

[6] T. Bienkowski. (2018, March) No sooner did the ink dry: 1.7 tbps DDoS attack makes history. [Online]. Available: https://www.netscout.com/blog/security-17tbps-ddos-attack-makes-history

[7] T. Vissers, T. Van Goethem, W. Joosen, and N. Nikiforakis, "Maneuvering around clouds: Bypassing cloud-based security providers," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 1530–1541.

[8] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu, "Portcullis: Protecting connection setup from denial-of-capability attacks," in *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '07. New York, NY, USA: ACM, 2007, pp. 289–300.

[9] X. Yang, D. Wetherall, and T. Anderson, "TVA: A DoS-limiting network architecture," in *IEEE/ACM Trans. Netw.*, vol. 16, no. 6, Dec. 2008, pp. 1267–1280.

[10] ——, "A DoS-limiting network architecture," in *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '05. New York, NY, USA: ACM, 2005, pp. 241–252.

[11] X. Liu, X. Yang, and Y. Xia, "Netfence: Preventing internet denial of service from inside out," in *Proceedings of the ACM SIGCOMM 2010 Conference*, ser. SIGCOMM '10. New York, NY, USA: ACM, 2010, pp. 255–266.

[12] X. Liu, X. Yang, and Y. Lu, "To filter or to authorize: Network-layer DoS defense against multimillion-node botnets," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08. New York, NY, USA: ACM, 2008, pp. 195–206.

[13] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker, "Controlling high bandwidth aggregates in the network," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 3, pp. 62–73, Jul. 2002.

[14] K. Whalen. (2017) The economics of DDoS attacks. Arbor Networks. [Online]. Available: https://www.arbornetworks.com/blog/insight/economics-ddos-attacks/

[15] W.-C. Feng, "The case for TCP/IP puzzles," in *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, ser. FDNA '03. New York, NY, USA: ACM, 2003, pp. 322–327.

[16] A. Juels and J. Brainard, "Client puzzles: A cryptographic countermeasure against connection depletion attacks," in *Proceedings of the 1999 Networks and Distributed System Security symposium (NDSS)*, March 1999.

[17] X. Wang and M. K. Reiter, "Defending against denial-of-service attacks with puzzle auctions," in *Proceedings of the 2003 Symposium on Security and Privacy*, May 2003, pp. 78–92.

[18] E. Nygren, S. Erb, A. Biryukov, D. Khovratovich, and A. Juels, "TLS client puzzles extension," Working Draft, IETF Secretariat, Internet-Draft, December 2016.

[19] Y. Nir and V. Smyslov, "Protecting Internet Key Exchange Protocol Version 2 (IKEv2) Implementations from Distributed Denial-of-Service Attacks," RFC 8019, 2016. [Online]. Available: https://rfc-editor.org/rfc/rfc8019.txt

[20] W. Feng, E. Kaiser, and A. Luu, "Design and implementation of network puzzles," in *Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.*, vol. 4, March 2005, pp. 2372–2382.

[21] X. Wang and M. K. Reiter, "Mitigating bandwidth-exhaustion attacks using congestion puzzles," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 257–267.

[22] D. Dean and A. Stubblefield, "Using client puzzles to protect TLS," in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10*, ser. SSYM'01. USENIX Association, 2001.

[23] T. Başar and R. Srikant, "Revenue-maximizing pricing and capacity expansion in a many-users regime," in *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Societies*, vol. 1, 2002, pp. 294–301.

[24] H. Shen and T. Başar, "Incentive-based pricing for network games with complete and incomplete information," in *Advances in Dynamic Game Theory: Numerical Methods, Algorithms, and Applications to Ecology and Economics*, S. Jørgensen, M. Quincampoix, and T. L. Vincent, Eds. Boston, MA: Birkhäuser Boston, 2007, pp. 431–458.

[25] H. Shen and T. Basar, "Optimal nonlinear pricing for a monopolistic network service provider with complete and incomplete information," in *IEEE Journal on Selected Areas in Communications*, vol. 25, no. 6, August 2007, pp. 1216–1223.

[26] D. Bernstein. (1997) SYN cookies. [Online]. Available: https://cr.yp.to/syncookies.html

[27] J. Postel, "Transmission control protocol," Internet Requests for Comments, RFC 793, September 1981. [Online]. Available: https://tools.ietf.org/html/rfc793

[28] W. Eddy, "TCP SYN flooding attacks and common mitigations," Internet Requests for Comments, RFC 4987, August 2007. [Online]. Available: https://tools.ietf.org/pdf/rfc4987.pdf

[29] J. Lemon, "Resisting SYN flood DoS attacks with a SYN cache," in *Proceedings of the 2002 BSD Conference (BSDC'02)*. Berkeley, CA, USA: USENIX Association, 2002, pp. 10–10.

[30] C. Dwork and M. Naor, "Pricing via processing or combatting junk mail," in *Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology*, ser. CRYPTO '92. London, UK, UK: Springer-Verlag, 1993, pp. 139–147. [Online]. Available: http://dl.acm.org/citation.cfm?id=646757.705669

[31] M. K. Franklin and D. Malkhi, "Auditable metering with lightweight security," in *Proceedings of the International Conference on Financial Cryptography*. Springer, 1997, pp. 151–160.

[32] T. J. McNevin, J.-M. Park, and R. Marchany, "pTCP: A client puzzle protocol for defending against resource exhaustion denial of service attacks." Department of Electrical and Computer Engineering, Virginia Tech, Tech. Rep. TR-ECE-04-10, October 2004.

[33] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[34] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "SoK: Research perspectives and challenges for bitcoin and cryptocurrencies," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 104–121.

[35] B. Laurie and R. Clayton, "Proof-of-work proves not to work; version 0.2," in *WEIS*, 2004.

[36] L. Chen, P. Morrissey, N. P. Smart, and B. Warinschi, "Security notions and generic constructions for client puzzles," in *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ser. ASIACRYPT '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 505–523.

[37] B. Groza and B. Warinschi, "Cryptographic puzzles and DoS resilience, revisited," *Des. Codes Cryptography*, vol. 73, no. 1, pp. 177–207, Oct. 2014.

[38] D. Stebila, L. Kuppusamy, J. Rangasamy, C. Boyd, and J. G. Nieto, "Stronger difficulty notions for client puzzles and denial-of-service-resistant protocols," in *Proceedings of the 11th International Conference on Topics in Cryptology: CT-RSA 2011*, ser. CT-RSA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 284–301.

[39] C. Sheng, "A general utility function for decision-making," in *Mathematical Modelling*, vol. 5, no. 4, 1984, pp. 265 – 274.

[40] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message syntax and routing," Internet Requests for Comments, RFC 7230, June 2014. [Online]. Available: https://tools.ietf.org/html/rfc7230

[41] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, 1993.
[42] ab - Apache HTTP server benchmarking tool. Apache. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html
[43] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab, "Design, deployment, and use of the DETER testbed," in *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test (DETER 2007)*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–1.
[44] C. Herley and P. C. van Oorschot, "SoK: Science, security and the elusive goal of security as a scientific pursuit," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP)*, May 2017, pp. 99–120.