Defending Against Denial-of-Service Attacks with Puzzle Auctions
(Extended Abstract)

XiaoFeng Wang* Michael K. Reiter†

Abstract

Although client puzzles represent a promising approach to defend against certain classes of denial-of-service attacks, several questions stand in the way of their deployment in practice: e.g., how to set the puzzle difficulty in the presence of an adversary with unknown computing power, and how to integrate the approach with existing mechanisms. In this paper, we attempt to address these questions with a new puzzle mechanism called the puzzle auction. *Our mechanism enables each client to "bid" for resources by tuning the difficulty of the puzzles it solves, and to adapt its bidding strategy in response to apparent attacks. We analyze the effectiveness of our auction mechanism and further demonstrate it using an implementation within the TCP protocol stack of the Linux kernel. Our implementation has several appealing properties. It effectively defends against SYN flooding attacks, is fully compatible with TCP, and even provides a degree of interoperability with clients with unmodified kernels: Even without a puzzle-solving kernel, a client still can connect to a puzzle auction server under attack (albeit less effectively than those with puzzle-solving kernels, and at the cost of additional server expense).*

1 Introduction

Denial-of-service (DOS) attacks aimed at exhausting a target server's resources have become a major threat to today's Internet. With the help of automatic attack tools (such as Tribal Flooding Network (TFN), TFN2K, Trinoo, and stacheldraht) [6, 10, 11, 17], not only can adversaries control large number of *zombie computers* to launch a distributed denial-of-service attack (DDoS), but they also might be able to simulate roughly normal traffic patterns on victim's network—in particular, with zombies using their authentic IP addresses. Such attacks pose challenges for existing countermeasures, such as intrusion detection, ingress filtering and TCP SYN-cookies.

Client puzzles [18, 2, 9] are an intriguing countermeasure which assist in defending against such attacks. This approach forces each client to solve a cryptographic "puzzle" for each service request before the server commits its resources, thereby imposing a large computational task on adversaries bent on generating legitimate service requests to consume server resources. Despite its promise, this approach has not enjoyed much use in practice, and we believe this is for at least two reasons. First, puzzles add to legitimate clients' load, and in the presence of adversaries with unknown computing power, it may be difficult to appropriately tune puzzle difficulty to minimize legitimate client cost. Second, very few implementations of client puzzles are available. This hinders us from studying the effectiveness of the idea in practical environments, its costs, and interoperability with existing protocols.

In this paper, we propose a new puzzle mechanism called a *puzzle auction*. Our auction lets each client determine the difficulty of the puzzle it solves and allocates server resources first to the client that solved the most difficult puzzle when the server is busy. This gives each client the flexibility to choose service priority against its valuation (computation paid for the service). We further design a bidding strategy for clients to raise the puzzle difficulty (bid) gradually via retransmissions to just above adversaries' capabilities. We describe our implementation of this idea within the TCP protocol in the Linux kernel. Our implementation achieves full compatibility with the original protocol, and even provides a degree of interoperability with clients having unmodified kernels: Even without a puzzle-solving kernel, a client still can connect to

*Department of Electrical and Computer Engineering, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA 15213, USA; xiaofeng@andrew.cmu.edu

†Department of Electrical and Computer Engineering and Department of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh PA 15213, USA; reiter@cmu.edu

the puzzle-auction server under attack (albeit less effectively than a modified client can and with greater server expense).

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes our puzzle auction model and analyzes its security characteristics. Section 4 presents an implementation of the model within the TCP protocol stack of the Linux kernel and empirical results with this implementation. Section 5 concludes the paper and discusses future work.

2 Related work

In this section, we first review countermeasures to DDoS attacks and then focus on the existing client puzzle techniques.

2.1 Countermeasures to denial-of-service attacks

The type of DoS attacks in which we are interested are attempts to consume the limited resources of a server, which include network bandwidth, CPU cycles, memory, disk space and network connectivity [7]. According to CERT [7], the most frequent attacks involve network connectivity, for example, the well-known *TCP SYN flooding* attack [8]. Throughout the paper, we take this attack as an example.

TCP SYN flooding aims to exploit a weakness in the TCP connection establishment protocol (three-way handshake) where a connection state can be *half-open*. A typical TCP three-way handshake proceeds as follows: A client first sends a SYN packet to the server. Upon receipt of the SYN, the server allocates state to hold information associated with the half-open connection and sends back a SYN-ACK packet to the client. The client completes the connection by replying with an ACK packet. In SYN flooding attacks, attackers initiate many SYN requests without sending ACK packets. This exhausts the server's half-open waiting queue and thus blocks a legitimate client's request from being serviced. Moreover, standard TCP will not time out a half-open connection until a certain number of SYN-ACK retransmissions have been made. This usually takes several to more than one hundred seconds. Therefore, even an attacker with a low-bandwidth communication channel (e.g., dial-up connection) might cripple a server on a fast network.

Some of the most frequently investigated countermeasures to DoS attacks include intrusion detection and ingress filtering. These are less effective, however, when adversaries can use zombies' authentic IP addresses. There are other methods designed specifically to defend against TCP SYN flooding. *Random early drop* randomly discards the half-open requests when the waiting queue becomes full [19]. This does not work well when the attacking throughput substantially exceeds legitimate clients' packet rates because in this case, the legitimate client's requests will often be dropped. Another countermeasure that is widely implemented is *TCP SYN-cookies* [19]. This approach removes the *SYN-RECEIVED* (i.e., half-open connection) state in the TCP protocol and hides authentication data (hashing connection parameters with a server secret) in the initial sequence number (ISN) of the SYN-ACK. Only upon receipt of the ACK packet carrying a sequence number that can be used to authenticate the connection parameters, which the server re-derives, does the server allocate the data structure for the connection.

Although this approach seemingly eliminates the target of SYN flooding attack, i.e., the half-open connection queue, it is essentially only an authentication scheme to prevent use of spoofed IP addresses. Adversaries who are capable of intercepting SYN-ACK packets from the server or capturing large numbers of zombies to send SYN packets with authentic IP addresses, still can flood the server's accept queue (i.e., the queue that contains the complete connections) with ACK packets. Another difficulty with SYN-cookies is compatibility: it cannot encode all service parameters into ACK packets and thus prevents clients from using certain TCP performance enhancements and transactional TCP [19]. More seriously, if the ACK packet is lost, the server cannot reconstruct the connection state or thus retransmit a SYN-ACK. This does not happen in the original protocol with the SYN-ACK retransmission mechanism [19]. Finally, the approach is designed specifically for TCP, i.e., to defend against SYN flooding in the TCP protocol. It is not clear how to generalize the idea to prevent attacks on other resources such as CPU cycles and bandwidth.

2.2 Client puzzles

An inherent weakness of today's Internet applications is that attackers may consume significant server resources at little cost. Client puzzles are a technique that strives to improve this situation: The client is required to commit computing resources before receiving resources. The idea can even help to defend against attacks with large numbers of zombie computers: A study shows that existing DDoS tools are carefully designed not to disrupt the zombie computers, so as to avoid alerting the machine owners of their

presence [15]. When client puzzles are used, zombie computers are required to commit computing resources during attacks. This may alert the owner to the attacker's use of this machine and motivate the owner to stop the attack.

To our knowledge, the first proposal for using client puzzles to defend against connection depletion attacks appears in [18]. Client puzzles have also been proposed to similarly protect authentication protocols against denial-of-service [2]. More generally, cryptographic puzzles have been employed in related fashions in the contexts of key agreement [20], defending against junk e-mail [12], creating digital time capsules [22], metering Web site usage [13], lotteries [16, 23] and fair exchange [23, 14, 5]. Whereas most puzzle proposals impose a number of computational steps to generate a solution, a recent "memory bound" alternative imposing memory accesses has been devised in an effort to impose similar puzzle solving delay even on clients with very different computational power [1]. Though here we will employ computational puzzles, we expect that memory-bound puzzles could work equally well in our context and will explore this in future work.

The only prior implementation of client puzzles reported in the literature, to our knowledge, was done in the context of TLS [9]. Although [18] postulates a TCP-based puzzle scheme, it does not report an implementation. In addition, this proposed implementation is incompatible with TCP in that a computer with this puzzle mechanism will not be able to communicate to the computer without the mechanism. The proposal we describe here offers better backward compatibility.

3 Puzzle auctions

In this section, we first describe the assumptions made in our research. We then present our puzzle auction mechanism and analyze its effectiveness in defending against a broad range of DoS attacks.

3.1 Attack model

In our research, we make a series of assumptions on the adversary's capability. Many of these assumptions are similar to those in [18], and generally are no stronger.

Assumption 1 *Adversaries cannot modify the packets between any legitimate client and the server.*

Attackers capable of tampering with packets can launch DoS attacks by simply destroying these packets. The same assumption is discussed in [18]. On the other hand, similar to [18], our mechanism still works well when attackers have only limited capability to interfere the communication between the legitimate clients and the server.

Assumption 2 *Under DoS attacks, the server is at least capable of rejecting the incoming packets and sending reply messages to the origins of these packets.*

Our approach cannot protect the server if adversaries can produce traffic sufficient to totally saturate the server's bandwidth. In other words, we require that the server at least be able to reject requests and send out packets. This seemingly limits the applications of our mechanism because many DDoS attacks are characterized by flooding the server with a sheer volume of service requests, for example, as in ICMP-echo floods. However, we envision that by coordinating multiple routers installed with our mechanism, we might be able to check the attack flows before they converge to the victim. We will revisit this point in the conclusion section.

Assumption 3 *Adversaries can perform IP spoofing, and can eavesdrop on all packets sent by the server. The adversaries can also coordinate their activities perfectly so as to take maximum advantage of their resources, i.e., as if all zombie computers constitute a single larger computer.*

The first two assumptions were also made in [18]. Notably, the assumption that the adversary can read packets sent by the server renders SYN-cookies less effective. We emphasize that the attacker may have potentially large computing resources and we make no assumptions to constrain its capability to integrate these resources in an attack.

3.2 Overview and rationale

For our purposes, a puzzle consists of two algorithms: one (possibly randomized) algorithm for generating "candidate solutions" and one deterministic algorithm for verifying whether a candidate solution is an acceptable solution. A *trial* is one sequence of (i) generating one candidate solution using the first algorithm and (ii) verifying it using the second algorithm. We presume that there is no more efficient way to generate an acceptable solution than repeatedly performing trials until the verification algorithm reports a success.

The resource allocation problem we consider can be characterized as a tuple $\langle C, A, S, V, R \rangle$, where: $C = \{c_{i=1,2,...}\}$ is a set of legitimate clients; $A = \{a_{i=1,2,...}\}$ is a set of adversaries; S denotes the server; and

$V : C \cup A \to \mathbb{N}$ is a *valuation function* indicating the maximum number of trials a client (either legitimate or malicious) is willing to perform to obtain the resource in a reasonable waiting period, assuming that it will actually obtain the resource having performed these trials.

The parameter R is a model $R = \langle L, \tau \rangle$ for the server's resources: The server keeps a buffer queue with the length L as resources and allocates a buffer to each request upon deciding to service the request. Every request carries a priority chosen by the client. When the buffer queue is full, the server can deprive a request with low priority of its buffer to make room for a request with higher priority. If a request keeps the buffer for a time interval no less than τ, we say the service (of that request) is complete. Otherwise, we say the service fails. This model is called *buffer model*.

The buffer model is sufficiently general to describe a range of service types with limited resources. To show this, we consider two general categories of resources distinguished by Qie, Pang and Peterson [21]: *renewable resources* such as CPU cycles and the bandwidth of network; and *non-renewable resources* such as processes, ports, TCP connection data structures and locks. First consider services on renewable resources, and let r be the available service rate and T_s be the maximum time the client can wait for the resource. Now consider a client's request that arrives at time T': If among all requests inside the system between T' and $T' + T_s$, the priority of the client's request is within top rT_s requests, then the request will be served before $T' + T_s$. In other words, the adversaries need to submit at least rT_s requests with higher priorities within T_s time units to prevent the request from being served. This situation is described by an instance of our buffer model with $L = rT_s$ and service time $\tau = T_s$. Holding a buffer in this queue for a period no less than τ is equivalent to acquisition of the renewable resource. In services on nonrenewable resources, we treat the length of the buffer queue L as the total resources the server has and τ as the service time. A client must equip its request with a priority high enough to obtain a buffer and keep it until the completion of the service, where the service is preemptive.[1]

In the buffer model, we use an *all-pay auction* to allocate the server's limited resources. In an all-pay auction, all bidders pay their bids before the auctioneer announces the winner. All the payments are forfeited, while only the winner gets the resources. In our mechanism, each client who requests resources is asked to solve a puzzle at a level of difficulty of the client's choosing and attach the solution to its service request. The server maps the difficulty of the puzzle to the priority with a non-decreasing function. When the buffer queue is full, the server uses incoming requests with higher priority to supplant ones with lower priority in the buffer queue. We call this a *puzzle auction*.

In the puzzle auction, a client c computes the maximum number of trials v_c it will perform for solving puzzles based on its valuation $V(c)$. For example,

$$v_c = V(c) \tag{1}$$

is an "optimistic" choice that client c might make if faced with no further information about the probability of gaining service having performed $V(c)$ trials. If c has additional information, however, it may tune v_c more carefully. For example, suppose the client knows a non-increasing *risk function* $P(.)$ that maps the number of trials performed to the empirical probabilities of failure in the auction when the server is heavily loaded. Such an estimate might be obtained from the client's experience during periods of apparently heavy load, or statistics over the data about winning bids during loaded periods disclosed by the server. With this material, the client c can compute its optimal investment v_c by weighing its profit from the service, $V(c) - v$, against the loss if it does not get the service, v.

$$v_c = \arg\max_v \{(1 - P(v))(V(c) - v) - P(v)v\} \tag{2}$$

Formula 2 describes a security management process: A client trades off its gain by obtaining service against the associated costs (the number of trials) to make an optimal decision on her security strategy. We call the optimal number trials v_c its *optimal valuation*. When all clients have the same (or similar) risk functions,[2] a client's chance to win an auction depends on her valuation.

Proposition 1 *In a puzzle auction, a client c's probability of obtaining resources is a non-decreasing function of its valuation $V(c)$.*

An interesting observation from a survey of existing DDoS tools is that these tools are generally designed not to disrupt the zombie computers [15], so as to not alert the owner to their presence. In a puzzle auction, this observation suggests that zombies' valuations may often be lower than legitimate clients'. That is, legitimate users will be more tolerant of their computers

[1] Here, we assume nonrenewable resources are preemptive. For example, during the establishment of TCP connections, the server can drop some half-open connections and release the buffers when the half-open queue is full.

[2] This happens if clients make the estimates according to the server's data or they have interacted with the server for a long time, provided that the empirical distribution converges.

committing resources to solving puzzles than owners of zombie computers will be (for the attacker's requests). If this is true, then according to Proposition 1, legitimate clients stand a better chance to win the auction.

3.3 Auction protocol

In order to detail our auction protocol, we begin by adopting a particular puzzle type. Here we employ a puzzle similar to that of [2], consisting of a "nonce" parameter N_s created by the server and a parameter N_c created by the client. A solution to this puzzle is a string X such that the first m bits of $h(N_s, N_c, X)$ are zeros, where h is a public hash function. We call m the *puzzle difficulty*. We presume that generating candidate values for X is of negligible computational cost, and so treat the verification of a candidate X (i.e., an application of h) as the cost of a trial.

In our mechanism, we take this puzzle construction because it allows a client to select the puzzle difficulty it solves. More specifically, many network protocols have retransmission mechanisms. We exploit this and the above puzzle formulation to design a bidding strategy for clients to complete the service with minimal computation. Specifically, a client can send its first request without solving any puzzle. If the request is declined, the client knows that the server may be under an attack. Thus, it solves a puzzle and resends a new request with the solution. If the request is declined again, the client further increments the puzzle difficulty in the next retransmission. This process continues until either the client completes the service or her optimal valuation v_c has been reached.

The auction protocol additionally employs the following notation:

- r_c: a service request from client $c \in C$.
- BF: the set of all service requests already in the buffer queue. $|BF| \leq L$.
- DIF: a function mapping each service request to the level of difficulty of the puzzle solution it contains. For a puzzle solution X in r_c, if the initial m bits of $h(N_s, N_c, X)$ are zeros, then $DIF(r_c) = m$. For notational simplicity, we overload the function and denote $DIF(X) = m$.
- CD: the target puzzle difficulty for the client's request.
- $INIT$: the client's initial target puzzle difficulty.
- $INCR$: the value by which the client increments the target puzzle difficulty for its request.
- v_c: maximum number of hash operations client c will perform for this service request (computed using Formula (1) or (2)).

1. **Client sends service request:**
   (a) Client c sets $CD = 0$ and $X = 0$; computes v_c; and generates a new client parameter N_c.
   (b) Client c does a bruteforce search of the puzzle solution with the difficulty level CD in the interval $X \in [0, v_c]$:
   While ($DIF(X) < CD$ and $X < v_c$)
   $X = X + 1$.
   If ($X = v_c$)
   exit and report failure.
   Client c constructs a request r_c containing N_c and X, and sends the request to S.
2. **Server allocates resources:**
   (a) Server S periodically checks the buffer queue to clear the requests from BF that have completed service.
   (b) On receipt of the client request r_c, server S checks the client parameter N_c in r_c:
   If (any $r' \in BF$ contains N_c)
   drop r_c and goto 2(c).
   Now server S checks BF:
   If ($|BF| < L$)
   insert r_c into BF
   else if ($\forall r' \in BF : DIF(r') \geq DIF(r_c)$)
   drop r_c and goto 2(c)
   else
   locate a request $r' \in BF$ with the lowest puzzle difficulty among all requests in BF, drop r', insert r_c and goto 2(c).
   (c) Server S sends to client $\hat{c}$ a notification of service failure which contains the current server nonce N_s, where $\hat{c}$ is the client whose request $r_{\hat{c}}$ has been dropped (if any) at 2(b).
3. **Upon receipt of a failure notification, client retransmits:**
   Client c extracts the server nonce N_s from the message, and increases its bid as necessary:
   If ($CD < INIT$)
   $CD = INIT$
   else
   $CD = CD + INCR$
   Goto Step 1(b).

In the protocol above, N_c and N_s play roles similar, but not identical, to nonce identifiers as often used in cryptographic protocols. N_s should change periodically, say every T time units, to limit the reuse of puzzles (and their solutions). N_c is constrained merely to not be in use simultaneously by two different requests. Though the adversary can consecutively reuse a puzzle solution X for the same N_c for up to T time units (i.e., before N_s changes), it still needs to generate at least L puzzle solutions of sufficient difficulty in time T to flood the server. In practice, enforcing that only *simultaneous* requests bear different values for N_c, as opposed to all requests, avoids the server needing to store a large list of previously seen nonce identifiers (which could itself pose a DoS opportunity).

The puzzle auction protocol above is efficient in the sense that the client can raise its bid just above the attacker's bids to win an auction. In other words, if the client wins the auction, it wins with the minimum expected computation for the given adversary.

3.4 Security analysis

In this section, we analyze the security of the proposed puzzle auction mechanism. We consider the following setting: An adversary with Z zombie computers is trying to attack the server S by denying the service over a buffer $R = \langle L, \tau \rangle$. Here, we consider $\tau \ll T$, where T is the duration of the "server nonce period" before N_s is changed, and for simplicity we consider one legitimate client, i.e., $|C| = 1$. Let ξ be the event that the legitimate client $c \in C$ cannot complete the service. The objective of the adversary is to maximize the probability $Pr\{\xi\}$.

For simplicity, we assume that the client c starts bidding at the beginning of a server nonce period, and consequently that the attackers competing with c must, as well. Let $(b_0, b_1, \cdots, b_n)$ be a sequence of bids. The client first bids b_0. If rejected, it continues to bid b_1 and so on. In total, it retries no more than $n+1$ times. In solving a puzzle, we call a hash operation a *hash step*. We further assume that each zombie and the client can perform hash steps at the same rate s. We call s the *step rate*.

We assume the hash function is a random function (i.e., *random oracle* [3]). That is, for each input, the hash function independently and randomly (with uniform distribution) maps it to an output in the image space. The only restriction is that the same input always yields the same output. In practice, a good candidate for random oracle is MD5 with its output truncated [3]. The random oracle model gives us a geometric random variable for the steps used to solve a puzzle. Specifically, to solve a puzzle with initial m zero bits, a hash step can be viewed as a Bernoulli experiment with a probability of 2^{-m} to succeed. Throughout the rest of the paper, we describe and analyze the puzzle auction mechanism with this model.

Let us first look at the adversary's bidding strategy. In Assumption 3, we assume that the adversary has perfect coordination among zombie computers. Therefore, we can view the attacker as a "super computer" whose computing power is equal to the sum of all zombie computers'. That is, the adversary can perform hash operations at the step rate Zs.

We say a client *set a bid to difficulty level* m if in solving a puzzle, the client bids with the first solution it found whose difficulty level is no less than m. In order for the adversary to cause this bid to be dropped, the adversary must compute L bids of difficulty (at least) m; we are interested in how long it will take the adversary to generate L such bids. Let χ_i^m be the random variable describing the number of steps for computing the i-th bid in all L bids set to difficulty m. $\chi_1^m, \cdots, \chi_L^m$ are i.i.d. random variables. When L is large, we can approximate the total steps for computing the L bids: $\sum_{i=1}^{L} \chi_i^m$ with expectation $E[\chi_i^m]L = 2^m L$. (During this time, the legitimate client can compute roughly $s' \approx \frac{2^m L}{Z}$ steps.)

To support the above approximation, we need to investigate the probability that the adversary takes fewer steps to compute the L puzzles. This is answered by Proposition 2.

Proposition 2 *The probability of solving no less than L puzzles with difficulty at least m in $2^{m-1}L$ steps is no more than $exp(-\frac{1}{6}L)$.*

Proposition 2 shows that the attacker's probability to set L bids to difficulty m within $2^{m-1}L$ steps drops exponentially w.r.t. the length of the buffer queue L. For example, if the server has $L = 1024$ buffers, the attacker's chance to overbid is less than e^{-170}.

Approximation Assumption: *With a sufficiently long buffer queue, we ignore the adversary's probability to set all L bids to m within $2^{m-1}L$ steps.*

On this basis, we estimate the upper bound of the attacker's probability to launch a DoS attack with the following theorem.

Theorem 3 *Under the Approximation Assumption, for legitimate client c with step rate s, service time τ and a bid sequence $(b_0, b_1, \cdots, b_n)$, the probability that the attacker can successfully prevent the client from*

completing service is

$$Pr\{\xi\} < (1-2^{-b_0})^{\frac{2^{b_0-1}L}{Z}-\tau s}\prod_{i=1}^{n}(1-2^{-b_i})^{\frac{(2^{b_i-1}-2^{b_{i-1}-1})L}{Z}} \quad (3)$$

Let us take TCP as an example. Suppose the length of the server's half-open queue is 1024. The attacker controls 1024 zombies. The step rate s is 1 hash operation per microsecond. The client's SYN request needs to be kept in the buffer for 250 microseconds for the ACK packet. Let the client's bid sequence be $(15, 16, 17, 18, 19)$. With Formula 3, the upper bound of the attacker's probability to launch an attack is 0.2248. When the number of zombies drops to 500, the probability becomes 0.0502. This is an interesting result: Just think about the SYN-cookies approach. When the number of zombies is close to the server's buffer size[3], they can easily exhaust all the buffers, while using authentic IP addresses. While with the puzzle scheme, the legitimate client still has a reasonably high probability to complete the service. Figure 1 further illustrates that the probability of attack increases with the number of zombies and decreases with the buffer size.

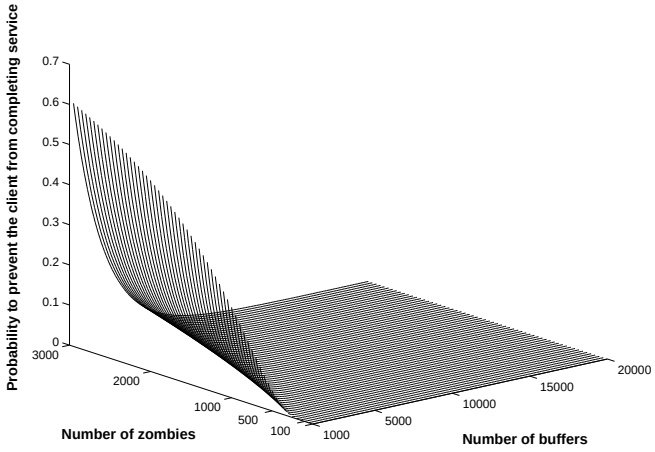

Figure 1. *Security of the puzzle auction.*

4 TCP puzzle auction

In this section, we describe our implementation of the puzzle auction mechanism in the TCP protocol stack of the Linux kernel, specifically version 2.4.17. Our implementation effectively defends against connection-depletion attacks on TCP, preserves compatibility with the original protocol and introduces only negligible overheads to the server. Our approach is also interoperable to a degree with clients having unmodified kernels: A client without a puzzle-solving kernel still has the chance to connect to a puzzle auction server under attack, albeit with less effectiveness and greater imposed server cost than a client with a puzzle-solving kernel.

In the following subsections, we first talk about how to construct the client puzzles with TCP service parameters and embed the auction mechanism into the three-way handshake protocol. We then elaborate on our implementation in Linux kernel, and finally present our experimental study.

4.1 TCP client puzzle

To embed our protocol into the TCP protocol stack, the first problem we need to solve is how to determine the client parameter N_c and the server parameter N_s. When establishing a TCP connection, the server decides whether a packet belongs to an existing connection or half-open connection according to its source IP address (SIP), destination IP address (DIP), source port (SP), destination port (DP) and the initial sequence number (ISN). In other words, the server does not allow two connections from the same client for the same ports and the same initial sequence number. Therefore, we can take these parameters, i.e., SIP, DIP, SP, DP and ISN, as the client parameter N_c. This treatment prevents clients from using the same puzzle to make two connections simultaneously. Moreover, it also simplifies the process to verify a puzzle: No extra work is necessary for detecting repeated client parameters because the existing classifier that filters incoming packets automatically does the job.

The server nonce N_s is supposed to change after each nonce period. A straightforward construction is to hash a server secret with a timer which increases for every nonce period. This guarantees that the server nonce changes periodically. Moreover, so as to make an adversary's task more difficult when it cannot eavesdrop on responses to requests bearing a spoofed IP address (in contrast to Assumption 3), we add the client's IP address to the input of the hash function for generating the server nonce. Thus, clients with different IP addresses are given different server nonces. If the adversary sends requests with spoofed IP addresses and cannot intercept the server responses, it will not obtain correct server nonces to compute solutions to puzzles. Figure 2 illustrates the construction of TCP puzzle, in which X represents the puzzle solution.

[3]Since a client may make more than one connection to the same service on the server, the adversary does not need 1024 zombies to circumvent SYN-cookies.

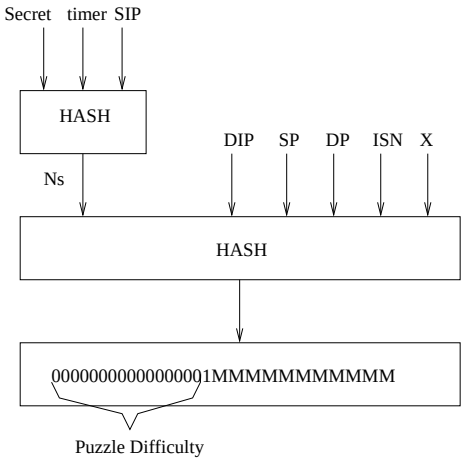

Figure 2. *TCP client puzzle.*

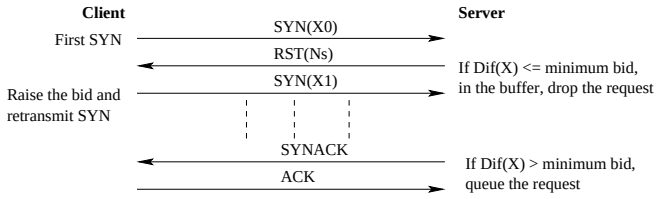

Figure 3. *TCP puzzle auction.*

To achieve compatibility, we advocate embedding the puzzle auction protocol into the communication flows of the original protocol. In practice, this is feasible because there are numerous *covert channels* in network protocols. In a TCP header, several fields can be used to carry the server nonce and the puzzle solution. During the three-way handshake, the SYN packet from the client has its acknowledgement sequence number empty, into which a 32-bit puzzle solution can be placed. We also take the RST packet as the failure notification and insert the server nonce N_s into its 32-bit sequence number field and, if a larger N_s is desired, in the window size and/or urgent pointer fields (for a total of up to 64 bits).

We roughly describe the TCP puzzle auction in Figure 3: A client first sends a SYN packet without a puzzle solution to the server. After receiving the packet, the server first checks the puzzle difficulty to determine the priority of the request (i.e., the difficulty of the puzzle solved, which should be small, since the client did not intentionally solve a puzzle), and then adds the request to the half-open queue if the buffer queue is not full. Otherwise, the server drops a request with the lowest priority (probably the new packet) and sends back a RST packet with the server nonce generated according to the client's IP address. The receiver of the RST packet uses the server nonce to increase its bid (i.e., compute a puzzle) and retransmits a new SYN with the puzzle solution. If the request is declined again, the client further raises its bid and does a retransmission again. This process continues until the client either receives the SYN-ACK or runs out the maximal number of retransmissions preset by the protocol.

The new protocol is backward compatible with the original protocol in the following sense. First, dropping the half-open connections is generally acceptable. Actually, many existing approaches (such as random early drop) do the same thing when the half-open queue is full [19]. Our protocol sends a RST packet to reset the session being dropped. This avoids the state inconsistency between the server and the client. Second, though the RST packet is not a part of the original three-way handshake protocol, it does not conflict with the original protocol: for the client not supporting puzzles, RST does nothing more than reset the current connection session. Finally, TCP requires that the client retransmit the SYN after exponential backoff once the connection timer expires. This repeats for several times, e.g., five times in Linux. Our protocol takes advantage of these retransmissions, especially the time interval of exponential backoff, to increase the difficulty level of the solved puzzle.

Generally, a drawback for client puzzles is that the client needs to install puzzle-solving software. If implemented within the network protocol stack, this may require modifying every client's kernel, which is generally not feasible. Our implementation, however, mitigates this problem: in the TCP puzzle auction protocol, the server determines the puzzle difficulty of a packet by computing $h(N_s, SIP, DIP, SP, DP, ISN, X)$. For a client without a puzzle-solving kernel, the puzzle solution X is fixed in each connection effort. However, it is still able to change the puzzle difficulty with different ISNs. TCP requires that each new session start with a more or less random ISN for preventing TCP hijacking [4]. A client can generate a new ISN by simply starting a new session. Our protocol supports launching new sessions consecutively by using a RST packet to immediately reset the client without a puzzle-solving kernel, thus saving it from doing exponential backoff. By resetting new sessions to query the server with different ISNs, a client will finally hit a puzzle difficulty high enough to complete a connection in most cases. This can be viewed as another strategy to solve puzzles that is undertaken by clients unaware of the puzzle mechanism: instead of performing hash operations itself, the client treats the server as an oracle to test its

solution. We call this strategy *bid and query.*[4]

In some sense, the bid and query strategy can provide a puzzle-solving interface to the application level. Instead of modifying the kernel, a small program is enough to keep querying the server, for example, a batch file or a Java applet helping to repeatedly try connections until one succeeds. Though not an ideal solution, this may be a useful temporary measure to permit the use of puzzle auctions even on client platforms where the network protocol implementation in the kernel cannot easily by modified (e.g., Windows). The bid-and-query approach allows for the puzzle-solving program to run effectively at the application layer.

A problem, however, is that the attackers also can take this strategy. More seriously, they do not need to wait for the answer of the query (SYN-ACK or RST) and thus can continuously generate numerous packets in hopes that many of them can get into the queue. However, this approach is limited by the server's bandwidth. For example, let us look at a 1 Gbps network. Since the shortest TCP/IP packet is 64 bytes, the maximal packet rate of this network is no more than 1,953,125 packets per second (pps). In the Linux TCP implementation, the server drops "old" half-open connections (i.e., that have timed out after sending the SYN-ACK at least once) when the queue is full. This may take only 9 seconds, during which the adversary can submit $1,953,125 \times 9$ packets. Since in expectation, only about $1/2^m$ of these will bear puzzles of difficulty at least m (supposing the adversary is choosing them randomly), if $L > (1,953,125 \times 9)/2^m$ then the adversary will probably fail to consume all L buffers with puzzles of difficulty at least m using this strategy. For example, if $L = 1024$ then a legitimate client will probably be able to succeed with a bid of only $m = 15$, which the legitimate client can generate in roughly 5 seconds using repeated queries to the server, assuming a round trip time (RTT) of 200 microseconds.

That said, if the attackers can generate such a large throughput, they do not need SYN flooding to attack the server, because they can already exhaust the server bandwidth. This is interesting because it is widely believed that SYN flooding needs a relatively small number of packets and thus is very easy to launch. Our TCP puzzle mechanism, however, raises the bar to this kind of threat and makes it potentially harder to exploit than bandwidth-exhaustion attacks.

[4]Semantically, a RST packet usually indicates to the client that no server is listening to a port, thus discouraging the client from reconnecting. To preserve these semantics for a client with a puzzle-solving kernel, a puzzle auction server can signal a dropped bid in some of the unused bits of the corresponding RST header. In the absence of this signal, the client can interpret the RST as meaning there is no server process listening on that port, as usual. A client without a puzzle-solving kernel, on the other hand, will attempt a connection for a preset number of times (without exponential backoff, and so this should proceed quickly) and stop after a number of tries without success.

4.2 Implementation

We built the puzzle auction mechanism into the IPv4 protocol stack in Linux kernel 2.4.17. We choose MD5 with its initial 64 bits as the hash function.

When implementing puzzles, an interesting question is precisely how to map the puzzle difficulty to the service priority. A straightforward solution is the direct mapping we described in previous sections. However, in practice, this approach is inefficient: Due to the randomness in puzzle solving time, an adversary can generate puzzles of varied difficulties within a fixed number of hash steps. In particular, even packets from an adversary not intentionally solving a puzzle at all may have a non-zero difficulty when evaluated at the server. This, in turn, can result in superfluous insertions and subsequent deletions from the half-open queue, each of which involves gaining mutual exclusion to the queue (since it is a critical region in the Linux kernel). Thus, excessive usage of the queue itself may cause a DoS. To minimize this risk, we map multiple puzzle difficulties to one priority. This reduces the server overhead but increases the legitimate client's computing costs for making a connection. At a high level, our strategy is to use a coarse-grained mapping for easy puzzles to reduce the server load and a fine-grained mapping for difficult puzzles to save the client's computation. In the implementation, we take the mapping in Table 1. In Appendix B, we also present a rough model and explain how to get this mapping table.

Puzzle Difficulty	Priority
0,1,2,3,4,5,6,7	0
8,9,10,11	1
12,13	2
$k > 13$	$k - 11$

Table 1. Mapping from level of puzzle difficulty to priority

Another question is how to structure the priority queue for puzzle auctions in the server. In the Linux TCP stack, there are two queues concerning the three-way handshake, the half-open queue and the accept queue. The half-open queue is organized as a hash table to accelerate the search for a half-open connection when its ACK packet comes. Removing the queue will reduce the system performance. In our implementa-

tion, we build a priority queue over the hash table. That is, a half-open connection is inserted in both the priority queue and the half-open queue. When a new request comes, the server checks it and drops the lowest priority request from the priority queue (and the corresponding entry in the half-open queue). When an ACK comes, the server locates its half-open connection from the half-open queue. For the convenience of maintenance, we adopted a doubly linked list as the data structure for both queues.

After a request completes the three-way handshake, the kernel moves it to the accept queue. Attackers also can flood this queue if they use authentic IP addresses or are capable of eavesdropping server outputs to other clients. In the Linux kernel, once the accept queue is full, not only are all requests in the half-open queue blocked from moving to the accept queue but also no new request is permitted to enter the half-open queue. To counter this threat, we also restructure the accept queue to be a priority queue. When this queue is saturated, the kernel resets the connection with the lowest priority and inserts the new one with the higher priority.

4.3 Experiment

In this section, we report our experimental study of the puzzle auction mechanism in a network environment. Our setup contains three computers: a client, a server and an attacker. The client is an obsolete Pentium Pro 199MHz machine with 64MB memory. The server has an Intel PIII/600 with 256MB memory. Both computers have a 2.4.17 kernel, either a standard one or a customized one with our puzzle auction mechanism. The attacker is strong, having two Intel PIII/1GHz CPUs and 1GB memory. It is also equipped with Linux kernel 2.4.17. Roughly speaking, the attacker has computing power ten times the client's. All these computers are attached to a 100Mbps campus network.

The objectives of this empirical study are: (1) evaluation of the overhead of the puzzle mechanism, (2) test of the performance of the system under SYN flooding attacks.

We first study the overhead of the puzzle auction mechanism. When a SYN packet is entering the system, the server first determines its priority. To avoid keeping too much information in the kernel, we configure the server to compute the server nonce for a request on the fly. Therefore, the server needs to take two hash operations to find out the priority of a request: one for the nonce and the other for the puzzle difficulty. The extra costs here are just a little bit above the two hash operations. In our experiment, we made 1,000 consecutive connections each to the server with standard kernel and our customized kernel. The standard kernel gave us the average connection time of 250.8 microseconds. The puzzle auction kernel had an average time of 255.4 microseconds. This empirical evaluation shows that the extra costs brought in by the puzzle auction mechanism are almost negligible: only 4.6 microseconds.

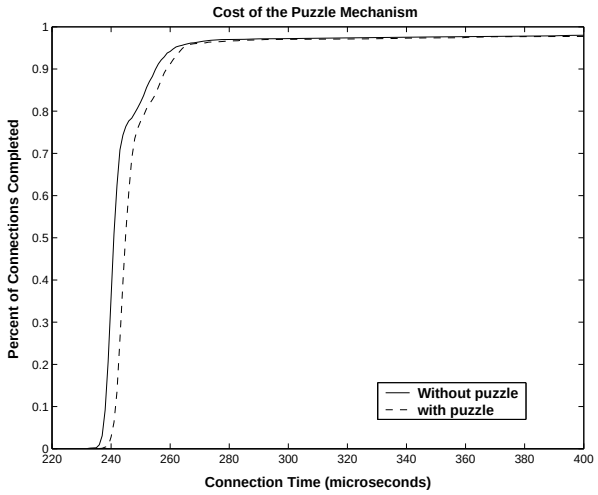

Figure 4. *Overheads of puzzle mechanism.*

The second experiment is on system performance in the presence of attackers with different computing power. In standard Linux, when a server is under a SYN flood attack, the kernel reduces the number of SYN-ACK retransmissions to two, so as to drop old requests quickly and make room for new requests. That is, a half-open connection may be held about 9 seconds (3 seconds for the first timeout and 6 seconds for the second). In our experiment, we first set the server's retransmission number under attack to 2 and then further reduced it to only one, which took about 3 seconds to discard a half-open connection after the half-open queue became full. We refer to the first server setting as *Setting 2* (i.e., two retransmissions) and the second as *Setting 1*. The server had a half-open queue with the buffer size of 1024 and followed Table 1 to map puzzle difficulty to priority.

On the attacker, we installed SYN flooding code capable of generating attack traffic with puzzles of varied difficulty levels. In the experiment, the attacker launched 5 attacks each on the server with different retransmission settings. These attacks set puzzle difficulty to 0, 8, 12, 14, 15. Without solving puzzles, the attacker started SYN flooding at a packet rate of 7,000pps. This rate easily brought down a server with the standard kernel. With the difficulty level of 8, the

attacker was still able to flood the server at the packet rate more than 5,000pps. However, its capability to generate packets was greatly impaired when the puzzle difficulty went above 12.

In the experiment, we tested two bidding strategies on the client: bid and query (BQ) and incremental bidding (IB). The BQ client had a standard kernel and a small program which made sequential and consecutive connections to the server. The IB client had a puzzle-solving kernel which automatically increased bids via retransmissions. Each experiment lasted until the client completed 500 connections successfully. After each successful connection, the client reset the connection and waited for a period randomly drawn from a uniform distribution between 0 and 150 milliseconds before trying again (so that the next attempt would not immediately reclaim the "opening" that closing the connection created). The *connection time* was measured to the point when the connection succeeds, and was restarted when the following connection attempt was initiated. The *average connection time* was computed by averaging the connection time over the 500 successful attempts.

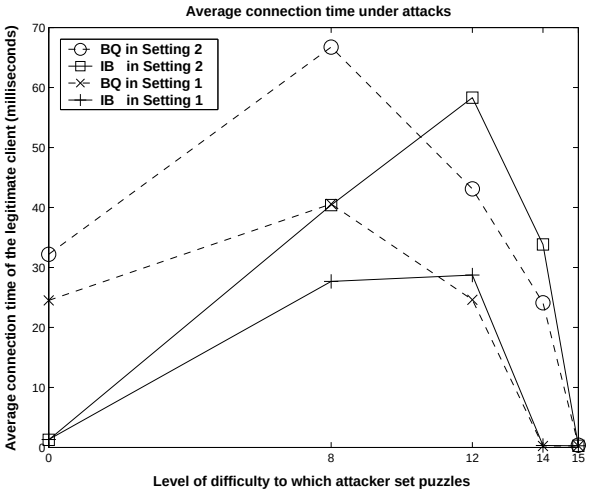

Figure 5. *Average connection time for BQ and IB w.r.t different retransmission settings in the server.*

Our experiment compared the average connection time of these two strategies in various attack and retransmission settings. The results are presented in Figure 5, where the X-axis indicates the difficulty level of the puzzles solved by the attacker and the Y-axis is the average connection time for the client. If the attacker did not solve puzzles at all, the client completed connections quickly: In either setting, the average connection time is around 30 milliseconds (ms) for BQ client and 1.3 ms for IB client. With the attacker's bids rising, the connection time was prolonged. For BQ client, the peak came when the attacker set the puzzle difficulty level to 8, 66.737 ms in Setting 2 and 40.581 ms in Setting 1. IB client, however, suffered the maximal delay at difficulty level 12, 58.308 ms in Setting 2 and 28.738 in Setting 1. The expected connection time dropped after the peaks, down to less than 500 microseconds at puzzle difficulty of 15. In general, the client performed well with the puzzle auction server. Even in the presence of strong attacker, a successful connection costs less than 0.1 second on the average.

It is perhaps surprising that the BQ client did well in the face of attacks with fairly difficult puzzles. An examination of the distribution of connection time reveals that the BQ client actually did not solely rely on the puzzles to make connections. Rather, it often captured the empty buffer left when the server dropped a half-open connection, and the following seems to be the explanation: With a standard kernel, after an attacker floods the buffer queue, the server will not accept new requests until the old half-open connections timeout. This constitutes a time interval during which no request is added or dropped. We call this a *holding period*. In a puzzle auction, however, the server still can update the queue (replacing the low priority requests with the high priority ones) throughout a holding period. In the experiment, owing to the random puzzle-solving time, the attack packets carried puzzles of various difficulty levels. This led to a continuous modification of the buffer queue, thereby spreading out the timeout deadlines of the requests inside the buffer. In other words, due to the "diffusion" of timeout deadlines, the server could get into a state in which there were requests timing out, say, every second. This phenomena became prominent especially when the attack throughput was high and priority granulation was fine. For example, with 5000 pps (difficulty 8) and Table 1 as the mapping function, the server would change 300 requests in the half-open queue within a second. As half-open connections were dropped, the greedy strategy of the BQ client was able to succeed relatively quickly. With a short round trip time (less than 200 microseconds), the client could complete the handshake quickly, before the attacker filled all the holes again. We call this phenomena *timeout diffusion*.

Figure 6 further illustrates the phenomenon. For the BQ client in Setting 1, its connection time is concentrated in the range from 10ms to 100ms when the attacker set its puzzle difficulty to 8. This amounts to making about 50 to 500 queries with a rough RTT of 200 microseconds. However, to solve a puzzle no easier than 12 (to beat the attacker bidding 8), the client

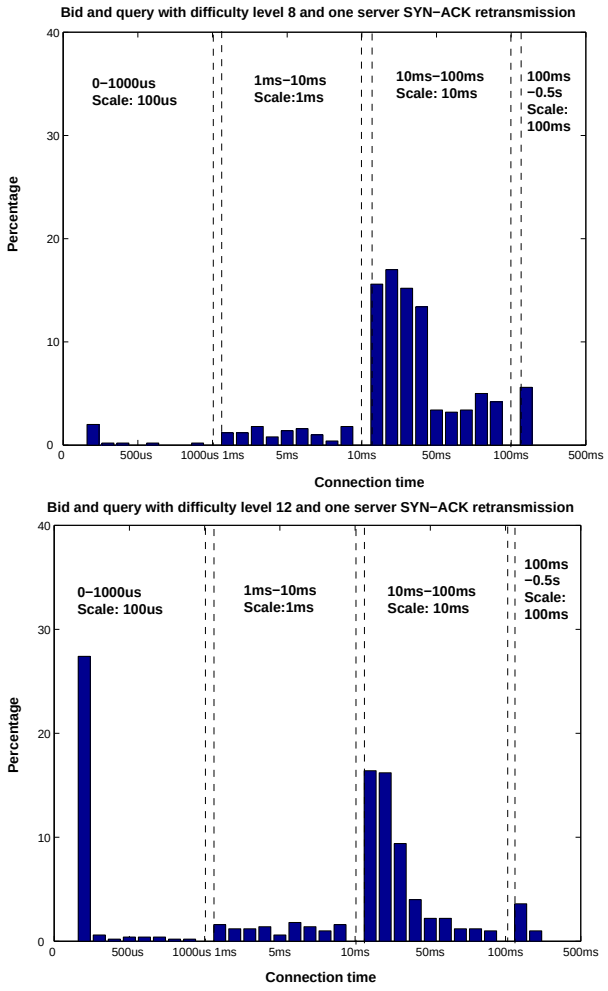

Figure 6. *Distribution of connection time for BQ client. Attacker puzzle difficulty: 8 (top) and 12 (bottom); Server retransmissions: 1.*

should try about 4,000 times on average. Therefore, the experimental results suggest that in many cases, the client kept querying the server until it captured a hole left by an old request timeout, and before it computed the right puzzle solution. When the attacker bid with a puzzle difficulty of 12, it could not keep the buffer full throughout a holding period. Thus, the BQ client got more chances: From Figure 6, we observe that nearly 30% of first attempts succeeded.

Timeout diffusion occurred due to the replacement of low priority requests with the high priority ones. This boosted the puzzle difficulty of the attacker's requests in the half-open queue. For example, when the attacker set its bids to a minimum difficulty level of 8, generating 5000pps produces about 300 puzzles per second with a difficulty level of 12. Since in Setting 2, the attacker had 9 seconds to update the puzzles in the buffer queue, it is probable that there was a time interval during which all bids in the buffer were at least 12. The IB client who requested the service in that interval would raise its bid to 14. The same problem happened at difficulty level 12 and thus gave the longest average connection time to IB client.

One solution to the problem is to shorten the holding period, thereby squeezing the attacker's time interval to accumulate difficult puzzles in the buffer queue. From Figure 5, we can see that in Setting 1, IB client did much better than in Setting 2. This is because the former kept requests for only 3 seconds, leaving the attacker a smaller chance to over bid. A shorter holding period also increases the frequency of dropping old requests, which benefits the BQ client, as well. In general, reducing the number of server retransmissions improves the system performance under attack. The cost, however, is the risk of inconsistency between the server and client states once the SYN-ACK is lost twice.

Although BQ performs comparably to IB, this largely owes to the short RTT in the experiment. Once the server sits outside the client's network, a RTT on the order of milliseconds will raise the connection time. In addition, a BQ client consumes significantly greater server resources. Thus, we emphasize that this should be considered at best a temporary approach to enable a client without a puzzle-solving kernel to participate in puzzle auctions.

5 Conclusions and future work

Client puzzles are a promising way to limit the adversary's capability to launch DDoS attacks. However, there are still many challenges to be addressed, especially, how to adjust puzzle difficulty in the presence of adversaries with unknown computing power, and how to implement it in practical settings. In this paper, we presented a puzzle auction and its implementation within TCP, as one way of addressing these challenges. Our mechanism allows clients to bid for service by computing puzzles with difficulty levels of their own choosing. The server under attack allocates its limited resources to requests carrying the highest priorities. We also designed a bidding strategy with which a client can gradually raise its bids until it wins. This minimizes its expected computing costs for obtaining resources even in the presence of adversaries with unknown computing power.

Our implementation of the auction mechanism in TCP takes advantage of covert channels in the TCP layer to achieve compatibility with the original protocol. Our experimental study shows that the imple-

mentation performed well under attacks launched by strong adversaries and incurred only negligible costs in the non-attack case. An interesting property of our approach is that it provides a way for clients without puzzle-solving kernels to participate in the auction, albeit with less effectiveness and greater cost to the server. In our implementation, a client with a standard kernel can connect to the puzzle auction server under attack through a small program running at the application layer. Of course, this approach does not scale, and so should be considered a temporary measure. In comparison to alternative techniques like SYN-cookies, we believe our approach is more effective against attackers with authentic IP addresses (or that can eavesdrop on server responses to IP addresses the attacker does not own).

Of course, client puzzles are an effective defense primarily when attackers have difficulty capturing vast computing resources. Interestingly, existing research projects on collecting idle CPU cycles on the Internet may bring into question the viability of this assumption. A potential way around this challenge is to replace our CPU-bound puzzle construction with memory-bound functions, for which the solution time should be far less variable as a function of the computing resources available to the attacker [1]. In our future research, we plan to examine this more closely.

In Section 3.2 we framed the puzzle auction mechanism over a general model for service and resources, so that any system that can be captured in the model might be protected by our mechanism. In future work we intend to examine other such systems, e.g., email systems in an effort to mitigate spam, or within routers to protect bandwidth. In particular, in Section 4.1 we mentioned that our mechanism cannot prevent attacks with an extremely large volume of service requests. Unfortunately, this is characteristic of many DDoS attacks. However, implementing puzzle auctions within the IP layer, in routers, might be a promising defense. Before an attack, potentially each server (or administrative system) announces the maximum throughput it can manage from each upstream router. These routers then admit packets toward the server according to this throughput limitation. Once the incoming packet rate exceeds the specification, the routers only forward the packets that bear solutions to the most difficult puzzles. This will help to check the attack flows before they converge on the server. In our future work, we intend to explore this direction and further study such coordination among routers.

Acknowledgements

Wang is supported by the DARPA OASIS program, the PASIS project at CMU and NSF grant IIS-0118767. Reiter is supported by the DARPA OASIS and FTN programs, and the NSF. We are grateful to the anonymous reviewers for their comments, and to Pradeep Khosla and Ramayya Krishnan for their encouragement.

References

[1] M. Abadi, M. Burrow, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, 2003.

[2] T. Aura, P. Nikander, and J. Leiwo. Dos-resistant authentication with client puzzles. In *Proceedings of the Cambridge Security Protocols Worshop 2000*. LNCS, Springer-Verlag, 2000.

[3] M. Bellare and P. Rogaway. Random oracle are practical: A paradigm for designing efficient protocols. In *Proceedings of First ACM Annual Conference on Computer and Communication Security*, 1993.

[4] S. Bellovin. Defending against sequence number attacks. In *RFC 1948*, May, 1996.

[5] D. Boneh and M. Naor. Timed commitments (extended abstract). In *Proceedings of Advances in Cryptology—CRYPTO'00*, volume 1880 of *Lecture Notes in Computer Science*, pages 236–254. Springer-Verlag, 2000.

[6] CERT. Computer emergency response team, CERT advisory ca-2001-01: Denial-of-service developments. In *http://staff.washington.edu/dittrich/misc/ddos*, 2000.

[7] CERT. Denial of service attacks. In *http://www.cert.org/tech_tips/denial_of_service.html*, June 4 2001.

[8] CERT. Advisory ca-96.21: Tcp syn flooding and IP spoofing attacks. September 24 1996.

[9] D. Dean and A. Stubblefield. Using client puzzles to protect TLS. In *Proceedings of 10th Annual USENIX Security Symposium*, 2001.

[10] S. Dietrich, N. Long, and D. Dittrich. Analyzing distributed denial of service attack tools: The shaft case. In *Proceedings of 14th Systems Administration Conference, LISA 2000*, 2000.

[11] D. Dittrich. Distributed denial of service (ddos) attacks/tools resource page. In *http://staff.washington.edu/dittrich/misc/ddos*, 2000.

[12] C. Dwork and M. Naor. Pricing via processing or combating junk mail. In E. Brickell, editor, *Proceedings of ADVANCES IN CRYPTOLOGY—CRYPTO 92*, volume 1328 of *Lecture Notes in Computer Science*, pages 139–147. Springer-Verlag, 1992.

[13] M. K. Franklin and D. Malkhi. Auditable metering with lightweight security. In R. Hirschfeld, editor, *Proceedings of Financial Cryptography 97 (FC 97)*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[14] J. A. Garay and M. Jakobsson. Timed release of standard digital signatures. In *Proceedings of Financial Cryptography*, 2002.

[15] X. Geng and A. Whinston. Defeating distributed denial of service attacks. *IEEE IT Professional*, 2(4):36–41, July—August 2000.

[16] D. M. Goldschlag and S. G. Stubblebine. Publically verifiable lotteries: Applications of delaying functions (extend abstract). In *Proceedings of Financial Cryptography*, 1998.

[17] K. Houle, G. Weaver, N. Long, and R. Thomas. Trends in denial of service attack technology. In *http://www.cert.org/archive/pdf/DoS_trends.pdf*, October 2001.

[18] A. Juels and J. Brainard. Client puzzle: A cryptographic defense against connection depletion attacks. In S. Kent, editor, *Proceedings of NDSS'99*, pages 151–165, 1999.

[19] J. Lemmon. Resisting syn flood dos attacks with a syn cache. In S. J. Leffler, editor, *Proceedings of BSDCon 2002*. USENIX, February 11-14, 2002.

[20] R. C. Merkle. Secure communications over insecure channels. 21:294–299, April 1978.

[21] X. Qie, R. Pang, and L. Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. In *Proceedings of the 5th OSDI Symposium*, December 2002.

[22] R. L. Rivest, A. Shamir, and D. Wagner. Time-lock puzzles and timed-release crypto. 10 March 1996.

[23] P. Syverson. Weakly secret bit commitment: Applications to lotteries and fair exchange. In *Proceedings of IEEE Computer Security Foundations Workshop*, 1998.

A Selected proofs

Proof for Proposition 1:

Let $c', c \in C$ be two clients such that $V(c') \leq V(c)$. Our objective is to show that the optimal valuation of c', $v_{c'}$, is not higher than the optimal valuation of the client c, v_c. The result obviously holds if v_c and $v_{c'}$ are computed by Formula (1), and so here we assume they are computed by (2).

First, note that $(1 - P(v))(V(c) - v) - P(v)v = (1 - P(v))V(c) - v$. We will show that any v between 0 and $v_{c'}$ gives the client c a profit no more than $v_{c'}$ does. Specifically, since $P(.)$ is a non-increasing function, we have for any $v \in [0, v_{c'}]$:

$$\begin{aligned}(1 - P(v))V(c) - v &= (1 - P(v))V(c') - v \\ &\quad + (1 - P(v))(V(c) - V(c')) \\ &\leq (1 - P(v_{c'}))V(c') - v_{c'} \\ &\quad + (1 - P(v_{c'}))(V(c) - V(c')) \\ &= (1 - P(v_{c'}))V(c) - v_{c'}\end{aligned}$$

Therefore, $v_{c'} \leq v_c$.

Let m be the minimal puzzle difficulty for winning an auction. Let $P_w(c)$ be the client c's probability to win within v_c hash operations. With the random oracle model, we have $P_w(c) = 1 - (1 - 2^{-m})^{v_c}$. This gives us $P_w(c') \leq P_w(c)$ if $V(c') \leq V(c)$. □

Proof for Proposition 2:
Our proof requires the following Hoeffding Bound:

Lemma 4 Hoeffding Bound *Let $X_1, X_2, \cdots, X_n$ be a set of independent, identically distributed random variables in [0,1], and let $X = \sum_i X_i$. Then:*

$$Pr\{X - E[X] \geq \epsilon E[X]\} \leq exp\left(-\frac{1}{3}\epsilon^2 E[X]\right)$$

Under the random oracle model, we can take steps for solving puzzles as a sequence of i.i.d. Bernoulli random variables, with probability 2^{-m} for the outcome 1, that is, found the solution to the puzzle; and probability $1-2^{-m}$ for 0. Let $\chi_1, \chi_2, \cdots$ be this random variable sequence. Therefore, we have that in $2^{m-1}L$ steps, the total number of puzzles being solved is $\chi = \sum_{i=1}^{2^{m-1}L} \chi_i$. With the linearity of expectation, it is easy to get $E[\chi] = L/2$. According to Lemma 4, we conclude:

$$Pr\{\chi - E[\chi] \geq E[\chi]\} \leq exp\left(-\frac{1}{3}E[\chi]\right) = exp\left(-\frac{1}{6}L\right)$$

□

Proof for Theorem 3:

To prevent the client from completing the service, the attacker must block all of c's $n+1$ bids. With the Approximation Assumption, we ignore the probability that the adversary generates L bids with difficulty level at least b_i within $2^{b_i-1}L$ steps. The total Z zombies allow the adversary to complete hash steps Z times faster than the client c. That means, if the client c can set its bid to difficulty b_i within $2^{b_i-1}L/Z - \tau s$ steps, the adversary cannot prevent the client to complete the service. On the other hand, from the adversary's viewpoint, this is the necessary condition to block the client. Let ξ_i be the event that the adversary successfully prevents c from getting the service after the client bids $b_0, \cdots, b_i$. We have the client's probability of failure in the first bid is:

$$Pr\{\xi_0\} < (1-2^{-b_0})^{\frac{2^{b_0-1}L}{Z}-\tau s}$$

The adversary can continue to block b_1 only if the client cannot generate the bid b_1 in the following $(2^{b_1-1}L/Z - \tau s) - (2^{b_0-1}L/Z - \tau s) = (2^{b_1-1} - 2^{b_0-1})L/Z$ steps. This gives us:

$$Pr\{\xi_1\} < (1-2^{-b_0})^{\frac{2^{b_0-1}L}{Z}-\tau s}(1-2^{-b_1})^{\frac{(2^{b_1-1}-2^{b_0-1})L}{Z}}$$

Following this line of reasoning, we can obtain the result of the theorem. □

B Mapping puzzle difficulty to priority

Here we present a rough model to construct the mapping from puzzle difficulty to the service priority.

Let d be the difficulty of a puzzle. We map the difficulty between d and $d+\eta-1$ to the same priority rank. Let C_{drop} and C_{add} be the costs to drop and add a connection request respectively. Let C_{MD5} be the cost of a MD5 operation and σ be the request rate (number of requests per second) of all legitimate clients. The objective of the server is to minimize both its own costs and legitimate clients' costs. However, the server might not always weigh these two types of costs equally because it may care more about its own costs than the client's costs. We capture this intuition with a ratio μ.

For puzzles set to the difficulty level m, on the average, half of them have difficulty levels at least $m+1$. Consider the worst situation where every packet with difficulty more than m causes an additional add-drop operation. If we group η difficulty levels starting from d (including d) instead of one level into one priority rank, the expected savings of extra operations is $\rho(2^{-1}-2^{-\eta})$, where ρ denotes the throughput (packets per second). On the other hand, legitimate clients, who wish to get into the system, have to spend additional $(2^\eta - 2)2^d$ MD5 operations on the average to win the competition. Putting these pieces together, along with the weight ratio, we can formally describe the server's objective:

$$\min_\eta\{\rho(2^{-\eta}-2^{-1})(C_{drop}+C_{add})\mu + \sigma(2^\eta-2)2^d C_{MD5}\}$$
$$s.t. \quad \eta \geq 1$$

Let $\gamma = \frac{(C_{drop}+C_{add})\mu}{C_{MD5}}$. By adapting the above formula, we know that the optimal η must satify:

$$\min_\eta\{\frac{\gamma\rho}{\sigma}(2^{-\eta}-2^{-1}) + 2^{\eta+d} - 2^{d+1}\}$$

Let $g(\eta) = \frac{\gamma\rho}{\sigma}2^{-\eta} + 2^{\eta+d}$. We can determine η according to the first derivative condition: $\frac{\delta g(\eta)}{\delta\eta} = 0$. This leads to

$$\eta = \max\{\frac{1}{2}(\log\frac{\gamma\rho}{\sigma} - d), 1\} \tag{4}$$

For example, suppose the server sets $\gamma = 16$. That means, given the $C_{add} = C_{drop} = C_{MD5}$, the server would rather have the client perform 8 extra hash operations than do one extra operation on the half-open queue. Let the expected attack throughput be $\rho = 4098$pps and legitimate client's requests come at the rate $\sigma = 1$. According to Formula 4, the mapping from the levels of puzzle difficulty to priorities is presented in Table 1 in Section 4.2.