

# pTCP: A Client Puzzle Protocol For Defending Against Resource Exhaustion Denial of Service Attacks

Timothy J. McNevin<sup>†</sup>, Jung-Min Park<sup>‡</sup>, and Randolph Marchany<sup>‡</sup>  
{tmcnevin, jungmin, marchany}@vt.edu

<sup>†</sup>Advanced Research in Information Assurance and Security (ARIAS) Lab

<sup>‡</sup>Virginia Tech Information Security Office

<sup>†</sup>Bradley Department of Electrical and Computer Engineering  
Virginia Polytechnic Institute and State University

*Abstract – Over the past few years, denial of service (DoS) attacks have become more of a threat than ever. DoS attacks are aimed at denying or degrading service for a legitimate user by exhausting the resources for a particular system. Client puzzle protocols have received attention in recent years as a method for combating DoS attacks. In a client puzzle protocol, the client is forced to solve a cryptographic puzzle before it can establish a connection with a remote server. This paper introduces a novel client puzzle protocol that utilizes a modification of the Extended Tiny Encryption Algorithm. An implementation of the client puzzle protocol was completed in the TCP stack of the Mandrake Linux 9.2 operating system. We call this modification to the TCP stack pTCP (for Puzzle TCP). Our client puzzle algorithm is very fast, and is portable to other systems and architectures. More importantly, it is very effective against connection depletion DoS attacks and other resource exhaustion DoS attacks (on the server) because minimal computation load is imposed on the server to verify the solution to a given puzzle. Our client puzzle protocol is also effective against various other resource exhaustion attacks within the transport layer, and can help prevent attacks that exist at the application layer. In this paper, we describe our client puzzle protocol in detail, and show its effectiveness against DoS attacks by using experimental results.*

**Keywords:** Denial of Service Attacks, Client Puzzles, TCP/IP, Tiny Encryption Algorithm.

## 1. INTRODUCTION

As the Internet becomes an integral part in many people's lives, the need to keep servers online and available has become increasingly important. In recent years, denial-of-service (DoS) attacks have become more sophisticated and effective at obstructing this availability. There are two different types of DoS attacks: local and remote [1]. Local DoS attacks are typically malicious software that denies or degrades service to the current user of the local computer. Remote DoS attacks, or network DoS attacks, can be defined as an attempt to disrupt a service on a remote computer by any means necessary to limit or deny that service to any client. A Distributed DoS attack (DDoS) is similar to a DoS attack, except that the number of attackers is increased and the location of the attackers is generally spread out across the topology of the Internet. The intended effect of this attack is to create more traffic that can exhaust resources at an end-host or a network router at a much faster rate. An effective attack would exhaust the resources of the router, and thus, preventing legitimate packets from reaching their destination. Attackers often compromise other computers, which are referred to as "zombies", and use them to perform this kind of attack. This method can subvert IP spoofing defense mechanisms and it also can decrease the probability of the attacker being detected. Despite the effectiveness of a large-scale DDoS, smaller DoS attacks are still prevalent throughout the Internet and they continue to disrupt services. The increasing sophistication and availability of these attack tools pose a major threat to the availability of the Internet and its associated services. Therefore, it is imperative that technologies to defend against these kinds of attacks are carefully studied, and eventually deployed to keep systems on the Internet available and running at all times.

Among the five layers of the Internet protocol stack [2], a DoS attack is usually only associated with three of the five layers: the application, transport, and network layer. In order to design the most effective countermeasure against DoS attacks, we must address the security shortcomings at each layer and introduce defense mechanisms that are capable of mitigating specific threats at the appropriate layer. In computer security, there is no "magical panacea" for all potential threats. The only way to defend a computer from attacks is to design and employ a number of protection mechanisms that are designed to mitigate a specific threat. The combined use of all of these mechanisms will provide the best protection against a wide range of attacks.

There are a large number of weaknesses in many applications within the application layer. It is difficult to summarize the type of attacks on the application layer because many software applications have their own custom protocols with unique features and potential vulnerabilities. Often the attacks are special in nature due to the specific design of each software

application. For this reason, DoS attack protection mechanisms need to be customized for each specific threat and type of application. To provide better security at the application layer, software engineers and designers need to make security a high priority when designing network applications.

To mitigate certain DoS or DDoS attacks at the network layer, the best solution is to provide an efficient and effective filtering mechanism that can distinguish between legitimate traffic and attack traffic [3, 4]. The major obstacle for preventing DoS attacks is the ability to determine if a packet belongs to an attacker or if it belongs to a real user. Filtering mechanisms at the network layer, or IP layer, are intended to create and detect attacker signatures within the IP packet by marking the packet at each router or by using existing elements within the IP header in order to determine if a packet belongs to an attacker or a real user [3, 4]. Many of the solutions that have been proposed are end-host filtering mechanisms. In this scheme, the end-host system will be responsible for storing and detecting the attacker signatures and will drop packets based on their knowledge. However, in a flooding attack, the objective of the attacker is to simply send a large number of packets towards the victim. In a large-scale flooding attack, despite the filter at the end-host, when a packet traverses through the network all the way to the victim, the goal of the attacker is essentially accomplished. An improved solution is to provide a distributive filtering mechanism that will be able to filter packets at various bottlenecks throughout the Internet. The ideal solution is to be able to filter packets as close to the attacker or zombie as possible.

In this paper, we focus on a defense mechanism for the transport layer, particularly for the Transmission Control Protocol (TCP) [5]. TCP is an end-to-end protocol that provides reliable data transmission in a connection-oriented fashion. Unlike the distributive filtering schemes for IP layer attacks, security mechanisms for the transport layer should be integrated into the end-to-end protocol. An example of an attack on the transport layer is the synflood attack [6, 7]. A synflood is accomplished when an attacker sends a large number of SYN packets to the victim, thus creating a large number of half-open connections that are stored on the server. The server has limited memory in terms of the number of half-open connections it can store. An attacker can effectively exhaust the server's resources by filling the buffer of half-open connections which denies service to future clients. These synflood attack programs are relatively easy to create, and are freely available online [6, 7].

Because a client puzzle protocol is inherently an end-to-end protocol, it can be readily implemented and integrated into TCP. For this reason, we argue that client puzzle protocols are a good DoS protection mechanism for the transport layer. We have designed and implemented such a protocol and named it pTCP. pTCP was created by modifying the three-way handshake of TCP. Before a client can establish a connection with a server, it must first solve a cryptographic puzzle. By forcing the client to solve this puzzle, we can prevent frivolous and abusive connection attempts by the client. pTCP is a fast and effective client puzzle protocol that is capable of preventing various forms of resource exhaustion DoS attacks by having a minimal puzzle generation time and by reducing the amount of computational load on the server for puzzle verifications. It is an effective solution to many DoS attacks because it can defend against synflood attacks and it can also mitigate problems that may arise at the application layer. The advantage to designing security mechanisms at lower layers within the network stack is that they may mitigate potential attacks on the higher layers [8]. By embedding a client puzzle protocol within the transport layer, we are able not only to defend against specific transport layer attacks, but also numerous and possibly unknown attacks that may exist at the application layer. In this paper, we present the design of a client puzzle protocol that uses a unique block-cipher based puzzle. We present the design details of our client puzzle protocol and verify its effectiveness through experimental results.

The rest of this paper is organized as follows: In the next section, we discuss some of the previous work on mitigating transport layer attacks. In Section 3, we introduce the client puzzle algorithm used in pTCP. In Section 4, we provide some of the implementation details of pTCP in the Linux kernel. We present our simulation results, and describe the performance of pTCP in Section 5. In Section 6, we discuss our plans for future research, and we conclude the paper in Section 7 by summarizing our contributions.

## 2. RELATED WORK

Over the past few years, several approaches have been proposed to prevent transport layer resource exhaustion attacks on systems, such as syncache, syncookies, and client puzzles.

Syncache [9] was designed to replace the linear chain of pending and incomplete connections. Syncache implements a global hash table that protects a server from resource depletion by limiting the size of the table and the amount of time spent searching for a pending connection. Despite this modification, syncache can still suffer from connection depletion attacks because a syncache bucket, or a hash chain in the hash table, can still overflow. Lemon [9] suggests that the syncookies mechanism should be used when this occurs.

Syncookies [9] focus on defending a server solely against a synflood attack. It accomplishes this by removing state information after sending the SYN-ACK packet to the potential client. The sequence number in the SYN-ACK packet is created by applying a hash algorithm on the client's information (destination port, source port, IP address, etc.) and a secret

key maintained by the server that changes every minute. The client increments this sequence number and sends an acknowledgement back to the server. The server decrements the number (which should yield the output of the hash function) and then reapplies the same hash function and compares the two numbers. If they are the same, the connection is established. This scheme requires no modification to the client's operating system. The syncookies scheme has a potential vulnerability if an attacker actually completes the TCP handshaking procedure. Such an attack can be carried out by zombies controlled by an attacker in a DDoS attack. The zombies would execute an extremely large number of TCP handshaking procedures with the server or maintain a large number of connections to exhaust resources of the server or of a particular application. Such an attack can be viewed as a "malicious" flash crowd (see Section 5.2.3 for more details on flash crowds). Although pTCP does not provide complete protection, our protocol does make it harder for the attacker to carry out such an attack. Because each zombie would need to solve a puzzle for each of the connection attempts, their computation load would increase dramatically. This means that the attacker would need to acquire a greater number of zombies or more powerful zombies to mount an effective DDoS attack against pTCP.

Juels and Brainard first introduced *client puzzles* to prevent connection depletion attacks [10]. Client puzzles were also devised to help prevent DoS attacks on authentication protocols [11]. Over the past few years, various client puzzle schemes have been proposed [12, 13, 14, 15]. The basic idea of a client puzzle is that when a server is under attack, it sends out a cryptographic puzzle for the client to solve before allocating resources for that client. The concept of cryptographic puzzles was first proposed by Merkle [16]. A cryptographic puzzle is created by taking a difficult problem from an appropriate cryptosystem and making it "feasible" by providing helpful information that will aid in finding the solution. This information reduces the solution search space so that the puzzle solver can simply apply brute-force techniques to find the correct solution. Therefore, as the name implies, puzzles are solvable problems that require the solver's time and effort. In a client puzzle protocol, a computational cost is invoked upon a client using a cryptographic puzzle, before granting the client access to particular resources on a remote system.

Since an attacker usually generates a large number of requests, it will have to solve a correspondingly large number of puzzles. In contrast, the legitimate client typically has only a small number of puzzles to solve. This method is effective in separating the attackers from the legitimate clients, and also gives the legitimate clients a better chance to connect with the server. In order for a client to prove that it is legitimate, it must use its own resources and time to show that it is not an attacker (by solving a puzzle). The ideal characteristics of a client puzzle protocol are summarized below [10, 11, 12]:

- First, a puzzle should be easy for the server to create and verify, and should be much more difficult for the client to solve. The level of difficulty can be parameterized, and can be changed if needed. However, if the server is not under an attack, it should be possible that a puzzle would not be generated at all, allowing the client access without solving a puzzle.
- Second, it should not be possible for an attacker to keep a table of known puzzles and solutions [10, 11].
- Third, the client should know that it has the correct answer before submitting it to the server. The puzzle solving process involves a repetitive brute-force task. The client should know when to terminate this process when it has the correct solution.
- Fourth, the server should know what puzzles it has generated and which ones to verify. There must be some type of mechanism in place that prevents an attacker from fabricating its own puzzle and sending its own solution to the server. The server needs to store a small amount of information so that it can determine which responses from the clients are solutions to valid puzzles.

The common problem among all of the client puzzle schemes that have been proposed is the puzzle verification, which involves the execution of a hash function or an encryption function to verify the client's answer. When creating a DoS resilient protocol, it is imperative that the defense mechanism itself does not become the basis for another DoS attack. An attacker can easily attempt to exhaust the computing power of a server by forcing it to verify a large number of incorrectly solved puzzles. In this scenario, the attacker does not bother solving the puzzles; it simply intends to force the server to waste its resources. Optimizing the puzzle verification mechanism is critical and doing so will undoubtedly improve the server's performance. pTCP is an optimized client puzzle protocol that strives to improve the server's performance while still maintaining its intended DoS resiliency.

In the vast majority of client puzzles that have been proposed, solving a puzzle involves reversing a one-way hash function by brute force. Depending on how the difficulty of the puzzle is set, the puzzle can be trivial or impossible to solve<sup>1</sup>. The puzzles presented in the related literature use different algorithms but they all require the client to solve the puzzle by brute-force [10, 11]. The entire reversal of a one-way hash function is considered computationally infeasible, but by revealing a portion of the answer to the client, the server creates a puzzle that is solvable. In hash-based puzzle algorithms,

---

<sup>1</sup> In general, adjusting the difficulty involves revealing varying degrees of a puzzle's solution.

the client has knowledge of an output value and a part of the corresponding input value to a hash function. Instead of attempting to reverse the hash function, the client solves the puzzle by using a brute-force method to find the rest of the input value. In a *server-generated puzzle*, the server generates the partial hash input and hash output, and transmits both pieces of information to the client. For every connection request, the server must execute the hash algorithm to create the puzzle and it must also execute the hash algorithm again to verify the client's solution to the puzzle.

There is another method for creating hash-based puzzles. In this method, the client is only given the partial hash input, and it needs to solve for both the rest of the hash input and the hash output [11]. We refer to this type of a puzzle as a *client-generated puzzle*. The difficulty level is set by forcing the first  $k$  bits of the output to be zero [11]. In this scheme, the server only needs to distribute a random number and the client will be responsible for creating the puzzle. In practice, a client-generated puzzle is better than a server-generated puzzle because the server can create puzzles much easier and faster. In a client-generated puzzle, the server needs to keep only two local variables (nonce and difficulty level), and respond with these values when a client makes a connection request. Furthermore, the server only needs to execute the encryption or hash algorithm once—when it is verifying a puzzle solution. On the other hand, in a server-generated client puzzle, the server must execute the encryption or hash algorithm twice for every connection request—once when issuing a puzzle and another when verifying the solution.

Wang and Reiter [12] present a client puzzle protocol, called a puzzle auction that was implemented and embedded into the TCP stack in Linux. The puzzle auction was designed to allow clients with a modified kernel to bid for a connection. The puzzle difficulty is the bid, and a higher bid implies a more difficult puzzle. In their incremental bidding scheme, the client bids for a connection by solving a puzzle at a certain difficulty and can re-bid for a connection by solving a puzzle of a higher difficulty and retransmit the request. Allowing a client to set the difficulty level may permit an attacker to control a zombie to purposely raise the difficulty level and to outbid other clients. The authors of [12] claim that most DDoS tools that execute on zombies are designed to operate quietly so that users will not notice their existence. They assume that solving puzzles repeatedly and of incremental difficulty will signal a user that their computer has been comprised. This is not always true, especially if the attack is launched during non-active times or using zombies where there is little human interaction with the computer. Another presumptuous assumption relevant to this issue is that the user of the zombie computer is a knowledgeable computer user and would take the appropriate actions when he/she realizes that his/her computer is being used as a zombie. Again, this is not always true. For the reasons stated above, the difficulty level of a puzzle scheme should always be controlled by the server. This issue is discussed further in Section 6.

For backward compatibility, the puzzle auction scheme [12] has a method for allowing a client with an unmodified kernel to complete a connection during an attack. Unfortunately, this implementation has potential vulnerabilities. In the implementation, a client with an unmodified kernel does not solve a puzzle. Instead, when the server receives a connection request, it is the server who computes the hash of a nonce, the source IP address, the source port, the destination IP address, the destination port, the initial sequence number, and another random value. If the output from the hash function meets the difficulty level, the connection is completed. A client can only establish a connection by repeatedly making attempts, and hope that the server can grant it. They call this scheme “Bid and Query”. This scheme clearly violates the first characteristic of an ideal client puzzle protocol, because the client and server are both required to perform a similar amount of work to complete the connection. This vulnerability can be taken advantage of by a malicious client.

Waters et al. [17] propose a new method to outsource client puzzles at the transport layer and briefly describe how their implementation could be modified for puzzles at the network layer. In their scheme, they claim that most clients do not wish to wait for a puzzle to be solved in order to complete a connection request. Instead, they introduce a complex method to solve puzzles before connection requests are actually made, thereby eliminating the time spent waiting for a puzzle to be solved. In their scheme, puzzles are solved in a certain time period, and then the puzzle answers are used in the following time period. The problem is that when a client first connects online it has to wait until the next time period to begin in order to use the answers it has computed from the last time period. The authors suggest that this time period be on the order of minutes and even use a time period of 20 minutes in their experiments. This means that a client would need to wait for 20 minutes before making a connection with a remote server.

In [18], a client puzzle protocol was designed and implemented at the network layer. While the concept of a client puzzle protocol at the network layer seems attractive, several critical issues need to be resolved before such an approach can be practical. The scheme described in [18] is not scalable, and it has several unresolved design issues. We will discuss more about network layer puzzle protocols in Section 6.

### 3. THE CLIENT PUZZLE ALGORITHM

The *Tiny Encryption algorithm (TEA)* is a block-cipher encryption algorithm that was proposed in 1994 by Wheeler and Needham [19, 20]. Both the encryption and decryption algorithms are Feistel type routines that encrypt or decrypt data by addition, subtraction, bit-shifting, and exclusive-OR operations. The goal of the encryption algorithm is to create as much

diffusion<sup>2</sup> as possible by incorporating many rounds or iterations of these operations. After TEA was released, certain minor weaknesses were discovered in the encryption algorithm [21]. In response, Wheeler and Needham developed an extension to TEA, called XTEA [22]. For our client puzzle algorithm, we use a variation of XTEA which uses 6 cycles. We call this variation *XTEA6*. In this encryption scheme, the plaintext is 64 bits long, the key is 128 bits long, and the ciphertext is 64 bits long. The encryption routine allows a parameter to be passed that indicates the number of iterations to execute in the main loop of the routine. We have selected 6 iterations because complete diffusion can be observed with that many rounds. Moreover, the level of security provided by 6 rounds is more than sufficient for a puzzle [22]. Note that in a puzzle algorithm, speed and efficiency are more important than robust security.

To the best of our knowledge, TEA was first suggested as a puzzle algorithm by Abadi et al. [15], although implementation of a client puzzle protocol using TEA was not attempted. As a client puzzle, XTEA6 has several advantages over most hash-based functions. As our simulation results will later show, XTEA6 is a much faster algorithm than most hash-based functions. This will allow a server to handle connections faster and will decrease the amount of waiting time on connections to complete for legitimate clients.

The client puzzle that we have developed is a client-generated puzzle. When the client requests a connection, the server responds with a server nonce,  $N_s$ , and the level of difficulty for the current puzzle. The server nonce is a 64-bit random number, generated with the Linux kernel random number generator. The server nonce is the same for every potential client and is changed every 60 seconds. The client uses the server nonce as a part of the puzzle that it must create. When the client receives the server nonce, it uses it as the plaintext in the XTEA6 encipher algorithm. The client must then find part of a 128-bit key value that will produce a particular ciphertext. More precisely, to solve a puzzle, the client needs to solve for the least significant 32 bits of the key. The rest of the key is comprised of the server's initial sequence number (ISN), the server port, the client's local port, and the client's IP address. By using the client's port and the server ISN, we guarantee that each puzzle will remain unique for each client. This concept was originally used in [12]. The difficulty of the puzzle is specified by the number of most significant contiguous bits in the ciphertext that must be zero. For example, if the difficulty level is  $k$ , then the  $k$  most significant bits of the ciphertext must be all zeros, the  $(k+1)$ -th most significant bit must be a one, and the rest of the bits can either be a one or a zero. Therefore, to solve a puzzle, the client needs to find a 32-bit portion of the key that will encrypt the given plaintext into a ciphertext that satisfies the requirement specified by the difficulty level. To save communication overhead, the client only sends back the 32-bit solution portion of the key to the server. Because the plaintext (i.e., server nonce), the difficulty level, and the remaining portion of the key are known to the server, the server only needs to receive the 32-bit value to verify the correctness of the puzzle solution. A graphical representation of the puzzle algorithm is shown in Figure 1. To compare our puzzle scheme with others, in addition to the XTEA6-based pTCP, we also implemented a version of pTCP that uses the MD5 hash function. In the MD5-based puzzle, the parameters and the size of the parameters are the same. A graphical representation of this MD5 puzzle scheme can be seen in Figure 2.

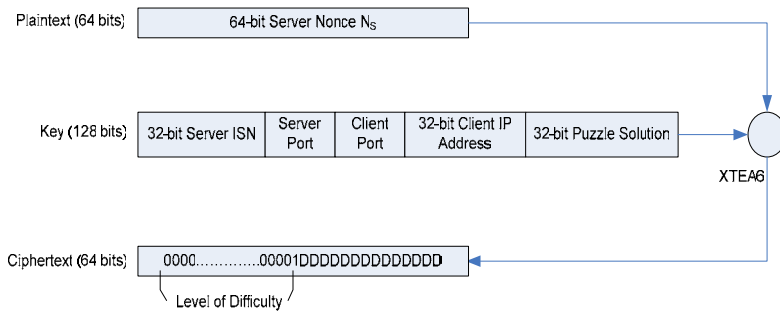


Fig. 1. XTEA6 client puzzle scheme.

#### 4. pTCP: A CLIENT PUZZLE PROTOCOL

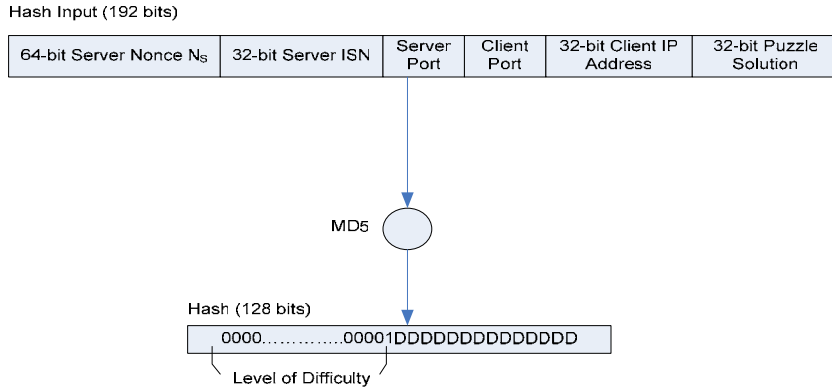
##### 4.1. OVERVIEW OF THE CLIENT PUZZLE PROTOCOL

In order for client puzzles to be truly effective, they must be placed below the application layer. Feng [23] documented the need to place client puzzles in the TCP or IP layer. In the following sections, we show how client puzzles can be

<sup>2</sup> In TEA, complete diffusion means that a single change in the plaintext propagates to 32 changes in the ciphertext.

integrated within the TCP stack to prevent resource exhaustion DoS attacks. This section also gives a detailed description on how the client and server interact in pTCP.

The current three-way handshake implemented in TCP has lead to security problems, mainly because the server can allocate resources before clients are authenticated. To prevent a client from depleting a server’s resources, it is necessary to modify TCP. pTCP is a modification to TCP that allows the server to issue a challenge to a potential client. Based on the number of pending active connections, the server should be able to determine if newer clients are required to solve a puzzle before establishing a connection. By placing the client puzzle protocol at the transport layer, we force a client to prove its legitimacy by solving a computational puzzle before a connection is made.



**Fig. 2.** MD5 client puzzle scheme.

pTCP implements a three-way handshake to establish a connection. When a server receives a packet with the SYN code bit set, it normally replies with a packet with the SYN and ACK code bits set (SYN-ACK packet). However, if the server is experiencing heavy traffic (e.g., flash crowds or DoS attack scenario), then it replies with a challenge to the client which includes a server nonce and difficulty level embedded into the SYN-ACK packet. A server can determine when this is necessary by examining the SYN queue. When the queue reaches near its maximum capacity, puzzles can be turned on. When a server issues a challenge to a client there are no resources allocated on the server, other than the server nonce and current difficulty level which is common to all potential clients. The server nonce is the same for all clients during a 60 second time period. If a client solves a puzzle in the previous time epoch and submits the answer in the following time epoch, the answer will be incorrect. The client’s connection will be reset and the client will need to make another attempt to complete the connection. Since most puzzles take only a few seconds to solve, we feel that having a 60 second server nonce period is adequate.

Unlike other client puzzle protocols, pTCP uses a client-generated puzzle, which means the server does not need to perform any cryptographic or hash operation for every potential client (SYN packet). When a challenge has been issued, the client parses the SYN-ACK packet for the server nonce and difficulty level, solves the puzzle, and replies with a solution in an ACK packet. The server then verifies the correct solution and then changes the state of the connection to “established”. Figure 3 illustrates how a client and a server communicate using pTCP.

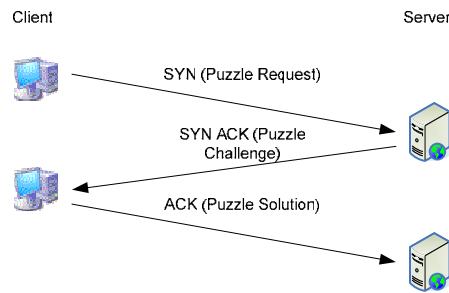
In pTCP, the server does not issue puzzle challenges during normal traffic conditions (which is indicated by the vacancies in the SYN queue). In this case, the server sends an acknowledgement (i.e., a normal SYN-ACK packet) to the client instead of a puzzle challenge. The client replies with an acknowledgement (i.e., a normal ACK packet). This flexibility of pTCP will allow clients with unmodified versions of TCP (i.e., TCP without puzzle challenge capability) to establish connections with servers using pTCP when the SYN queue level is sufficiently low. This is an important issue for backwards compatibility and client puzzle implementation in the Internet. In practice, the backward compatibility of a client puzzle protocol is very important because these protocols require changes in the clients’ software. A smooth migration from standard TCP to puzzle capable TCP will require the protocol’s backward compatibility.

A problem common to both pTCP and the syncookies scheme is the incapability to retransmit the SYN-ACK packet. Since all state information is removed on the server following the transmission of this packet, retransmissions are not possible in either scheme.

4.2. IMPLEMENTATION OF THE PROTOCOL INSIDE THE LINUX KERNEL

For our experiments, we implemented pTCP into the TCP stack in Linux [24]. In pTCP, all of the puzzle information is passed using the TCP header. Since data is not passed in the normal three-way handshake in TCP, any additional data need

to be placed within the TCP header. The options field in the TCP header was utilized to pass the puzzle requests, puzzle challenges, and puzzle answers. Since the options field is of variable length, many different options can be placed within this field at the same time.



**Fig. 3.** Client-Server interaction in pTCP.

The three types of options were given code numbers to avoid conflicting with other known TCP options. The puzzle request, puzzle challenge, and puzzle answer were given codes 99, 100, and 101, respectively. In pTCP, every client sends a puzzle request embedded into the initial SYN packet. If a server supports pTCP, this request will act as a signal that this client is capable of solving puzzles. Code was modified in the `tcp_transmit_skb()` and the `tcp_syn_build_options()` functions to add this information into the TCP header. The contents of the initial SYN packet can be seen in Figure 4a. When this packet is sent to the server, the server calls the function `tcp_parse_options()`. This function is responsible for parsing all of the options within the TCP header. After parsing the options, the server is ready to reply with a SYN-ACK packet. If the server determines that it is currently under a heavy amount of traffic, by examining the length of the SYN queue, it replies with a challenge. The `tcp_syn_build_options()` and `tcp_transmit_skb()` are called again to build the header and transmit.

In a SYN-ACK challenge packet, the current puzzle difficulty level and the current server nonce are both embedded into the options field in the header. The contents of a SYN-ACK challenge packet can be seen in Figure 4b. Following the transmission of a SYN-ACK challenge packet, the server removes the client information from memory, in the exact same way that syncookies removes client information from memory. Therefore, pTCP is resilient to synflood DoS attacks because no state information is stored by the server for pending connections.

The client, after having received the challenge packet, calls `tcp_parse_options()` and determines that a puzzle needs to be solved. The client uses the 64-bit server nonce it has just received, and uses that value as the plaintext for the XTEA6 encipher algorithm. The client then examines the TCP and IP headers for the server ISN, the server port and the local port, and the local IP address. It uses these values as part of the key used for the XTEA6 algorithm. The pseudocode for the puzzle solving function is shown in Figure 5.

When the client has successfully solved the puzzle, it creates an ACK packet and embeds the puzzle solution into the options field, as seen in Figure 6. When the server receives this packet, it is an ACK packet from an unknown client. The server calls a function to verify the puzzle solution. This function involves calling the XTEA6 encryption function only once. If the puzzle solution is correct, the state of the connection is changed to established, thus bypassing the half-open connection state.

Since the server accepts anonymous ACK packets with puzzle solutions, this can result in another type of an attack. An attacker could be sniffing packets and discover a solution to the puzzle that another computer has already solved. The attacker could then send this puzzle solution as its own. To counteract this, our puzzle uses various parameters from the TCP and IP header. By embedding these values into the puzzle, an attacker cannot use previous solutions.

Potentially, an attacker could flood the server with false puzzle solutions. This could potentially be another form of a DoS attack that attempts to exhaust the server's CPU power. Since the client puzzle algorithm is extremely fast, pTCP would be able to discard these false answers quickly and efficiently. It is important to always assume that attackers will modify their attack to exploit weaknesses in a modified version of TCP. To avoid such attacks, we chose the puzzle algorithm to be XTEA6 and use a client-generated puzzle scheme.

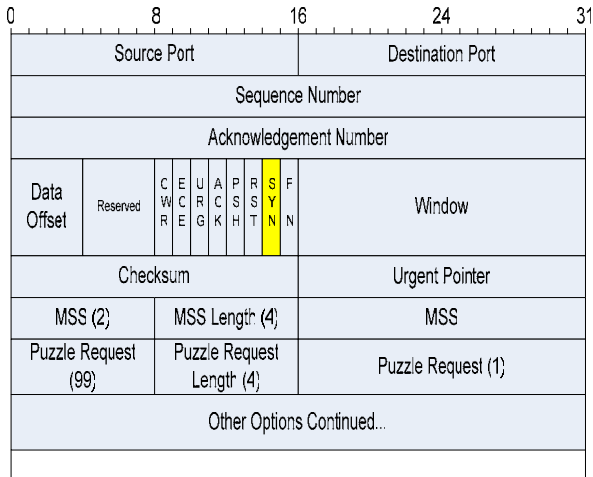


Fig. 4a. Puzzle request packet.

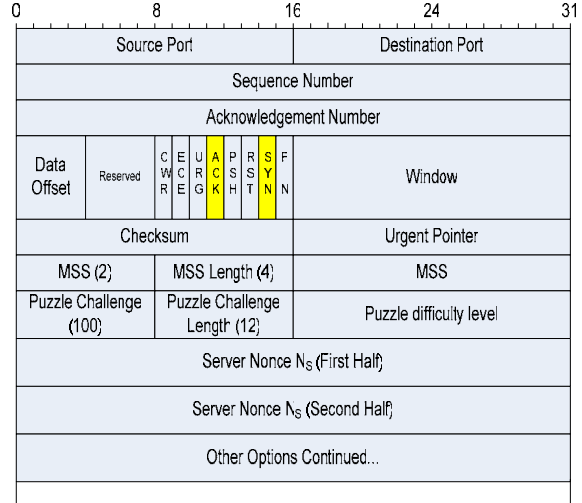


Fig. 4b. Puzzle challenge packet.

```

Plaintext =  $N_S$ 
Key[0] = Server ISN
Key[1] = Server Port || Local Port
Key[2] = Local IP address
While (Answer is not found)
    Key[3] = get_random_bytes( )
    Ciphertext = XTEA6(Plaintext, Key)
    If Ciphertext meets difficulty constraint
        Return
    Else
        Continue
    
```

Fig. 5. Pseudocode for puzzle solving algorithm.

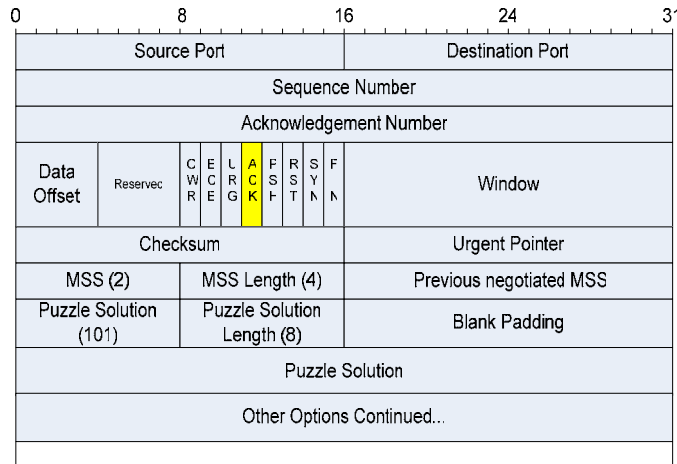


Fig. 6. Puzzle solution packet.

## 5. SIMULATION RESULTS

### 5.1. THE PUZZLE ALGORITHM

An important aspect to pTCP is the selection of the cryptographic algorithm used for the client puzzle. We compared the solution verification time of three puzzle algorithms—one uses XTEA6, and the other two use MD5 [25] and SHA-1 [26], respectively. According to our simulation results, the puzzle algorithm based on XTEA6 has the fastest solution verification time among the three that were examined.

To directly show how XTEA6 can improve the server’s solution verification performance, we measured the amount of time spent verifying puzzle solutions within the Linux kernel. Our comparison of XTEA6 pTCP and MD5 pTCP is shown in Figure 7. These results show the major advantages of using XTEA6 instead of MD5. The MD5 puzzle scheme can verify 1000 puzzles in around 31,000 microseconds. Meanwhile, an XTEA6 puzzle scheme can verify 1000 puzzles in less than 4,000 microseconds. Since pTCP uses XTEA6 it is far more efficient because it can verify a large number of puzzles much faster than any other puzzle scheme. When traffic is heavy, pTCP can improve load conditions on the server which can in return reduce the connection times for clients. From the results in Figure 7, XTEA6 is the best choice for a client puzzle because it will increase performance on the server since the puzzle verification time is considerably lower than an MD5-based puzzle scheme. It should be noted that the solve time for the XTEA6-based puzzle could be faster than that of puzzles based on MD5 or SHA-1 (assuming the difficulty level are the same for all three algorithms). This implies that an attacker may solve the XTEA6-based puzzle faster. This fact should be considered by a server when setting the difficulty level of the puzzles.

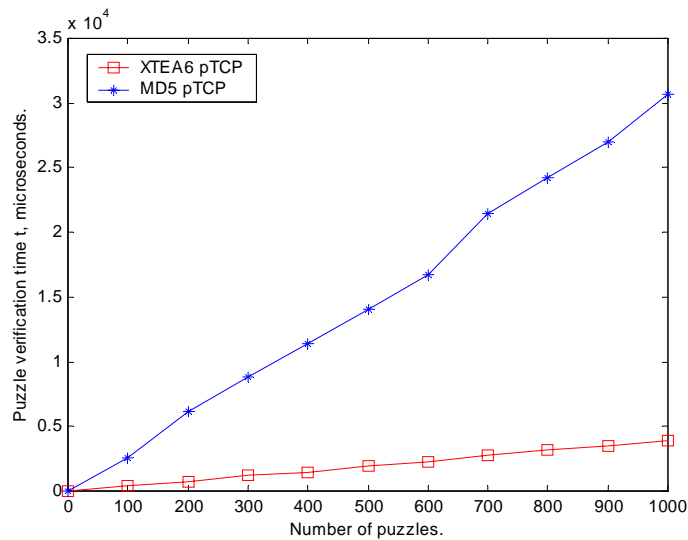


Fig. 7. Puzzle verification times for pTCP.

### 5.2. THE PERFORMANCE OF pTCP

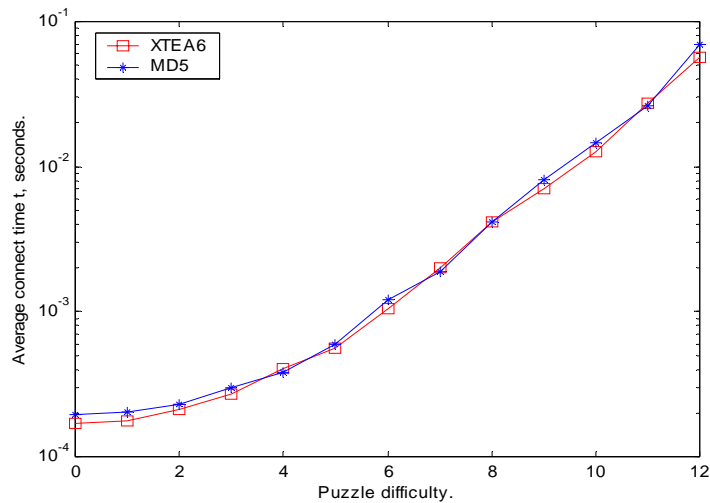
In this section we describe more of our simulation results with pTCP. Our first experiment varies the puzzle difficulty level and examines the connection times for MD5 pTCP and XTEA6 pTCP. Our second experiment measures the performance of pTCP during a synflood attack by measuring the connection times for a legitimate client. Our third experiment deals with flash-crowd scenarios, where there is a large amount of legitimate traffic that is being directed towards the server.

#### 5.2.1. MODULATION OF THE PUZZLE DIFFICULTY

In this simulation experiment, we created an environment that consisted of a server, a legitimate client, and an attacker. This environment will help determine the performance of pTCP and how it handles various attack scenarios. For our first simulation, the attacker executes 16 instances of a synflood attack program called synk4 [7]. In the standard TCP protocol, it created a sufficient amount of SYN packets to carry out a successful DoS attack. During this attack, a legitimate

client could rarely complete a connection, if at all. With pTCP, a large number of half-open connections will signal the server to begin distributing puzzles.

In pTCP, an important quantity to measure is the amount of time needed to establish a connection versus the puzzle difficulty level. It is important to verify that increasing the difficulty level increases the puzzle solve time, which in turn increases the connection time. For each connection, the client is solving a puzzle and following through with the three-way handshake. In Figure 8, we show the average connection time versus the puzzle difficulty during a synflood attack. In the figure, one can observe that the client’s connection time increases as the puzzle difficulty level is increased. This result verifies the fact that raising the difficulty level makes the puzzles more difficult, thus throttling the client’s connection attempts.



**Fig. 8.** pTCP connection time versus puzzle difficulty.

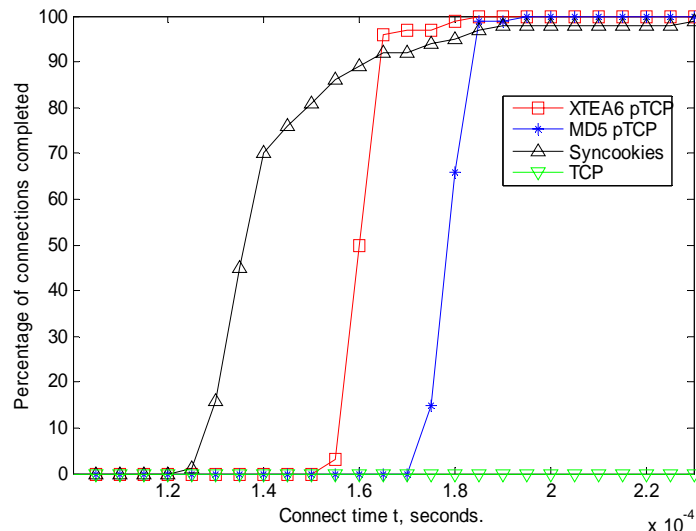
### 5.2.2. PERFORMANCE OF PTCP DURING A SYNFLOOD ATTACK

In order to test the effectiveness of pTCP during a synflood attack, we compared the protocol against two versions of the TCP protocol—one with syncookies turned on and the other with syncookies turned off. We also tested both versions of pTCP (puzzles based on MD5 and XTEA6).

The same simulation environment from the previous experiment was used again to test the performance of pTCP during a synflood attack. The attacker again executed 16 instances of the same synflood attack program. The `tcp_syn_max_backlog`<sup>3</sup> parameter on the server was set to the default size of 1024. The legitimate client made a large number of connections and we measured the amount of time it took to complete the connection. Our simulation results from this experiment are shown in Figure 9. In standard TCP with syncookies turned off, a legitimate client could not establish a connection with the server at all; every connection attempt had timed out after 5 retries, which from our observation is roughly 188 seconds. However, as expected, with standard TCP with syncookies turned on, every connection was completed. Recall that the syncookies scheme was designed to solely defend against synflood attacks. In our next experiment, under the same attack conditions, the server and the legitimate client both used pTCP, with the difficulty level set to zero. Due to the puzzle solve time, the puzzle verification time, and the increase in packet size, the connection times for pTCP were slightly greater than the connection times for syncookies. However, we noticed that XTEA6 pTCP was very comparable to syncookies because roughly 90 percent of the connections were completed by the same amount of time. The time difference between XTEA6 pTCP and syncookies for a connection request to complete is less than 30 microseconds. This small difference is due to the extra data being sent in each packet. The results show that pTCP can be effective in defending against synflood attacks and could be considered as an alternative to syncookies. The fact that pTCP can process TCP options (such as the maximum segment size) better than syncookies is another important advantage to pTCP. These TCP options are important for increasing performance for certain clients and servers where there is a larger amount of bandwidth. The inability to take full advantage of these options can have an impact on the connection speeds throughout the client’s entire session with the server. Having a mechanism that can prevent these attacks and still be able to recover the actions from the three-way handshake is the best choice for designing a modification to this protocol. Syncookies could be

<sup>3</sup> The `tcp_syn_max_backlog` parameter specifies the maximum number of half-open connections the server can store.

fixed to resend the options that were lost in the three-way handshake, but it was not done, since it would require a change to the client. We feel that if the operating system of the client is to be modified, it would be more appropriate to use pTCP because it can protect the server from more attacks.



**Fig. 9.** Percentage of connections completed versus connection times for pTCP, syncookies, and TCP.

### 5.2.3. PERFORMANCE OF pTCP DURING A FLASH-CROWD SCENARIO

A flash-crowd is defined as a dramatic increase in traffic to a Web site that results in the Web site being unreachable for a certain amount of time [27]. In a flash-crowd, the traffic is legitimate and does not belong to any attack. Unfortunately, a flash-crowd has the same effect as a DoS attack. A large amount of connections are created simultaneously and it may be possible for future clients to be unable to establish a connection. To prevent this, a client puzzle protocol could be deployed to throttle users' connection attempts so that the server itself is not overwhelmed. For our next experiment, we test the performance of pTCP during this scenario by using the tool `httperf` [28], which was used to send a large number of connection requests.

During a large surge of normal legitimate client traffic, the server can become overwhelmed and may not be able to respond to every client. A server equipped with syncookies will have no protection to defend against this type of an attack (whether it is unintentional or intentional). In contrast, pTCP can defend against synfloods and has the potential to improve server performance during flash-crowds. By shifting the computational load on to the client, the attacker will be forced to expend much greater amounts of computing resources to mount a resource exhaustion DoS attack on the server. In a DDoS attack, where an attacker is controlling a large number of zombies, the attacker would need to control more zombies to increase its total computing power and to increase the intensity of the overall attack. It should be noted that pTCP can only defend against end-host resource exhaustion DDoS attacks; it was not designed to mitigate resource exhaustion attacks targeted at the intermediate routers. For our experiments, we measured the average system load of both the client and server to verify that pTCP is able to shift the workload from the server to the client. The average system load can be roughly defined as the sum of the run queue length and the number of jobs currently running on the CPU [29]. In Linux, the average system load for the past minute, 5 minutes, and 15 minutes can be examined by calling the `uptime` command. The load average for the previous 1-minute time period is a moving average that is defined in Equation 1, where  $n$  is equal to the number of current active processes [29].

$$load(t) = load(t-1)e^{-5/60} + n(1 - e^{-5/60}) \quad (1)$$

Since the puzzle solving functionality is within the code of the Linux kernel, the more time spent solving a puzzle, the more jobs will be waiting to be executed, which implies a higher system load. According to [29], when the system load reaches 3 for a single processor computer, the computer is noticeably slower and overwhelmed.

One of the main goals of pTCP is to reduce the load on the server, thus, allowing it to handle more connections and improve overall system performance. For this experiment, we created another network test bed that consisted of 4 computers: 1 server and 3 traffic generating clients. The clients used `httperf` to repeatedly make connection requests (100 requests per second). The purpose of this experiment was to create a large amount of legitimate traffic to simulate a flash-

crowd scenario. We observed the average system load for the clients and the server while using standard TCP with syncookies enabled. We then observed the average system load for the clients and the server while using XTEA6 pTCP and then varied the puzzle difficulty level. We compared pTCP with syncookies, but the syncookies load measurements are constant. Figure 10a and 10b show the system load of the client and server, respectively. One can observe that when the puzzle difficulty is increased, the system load for the client increases because each client is solving a large number of puzzles. In Figure 10b, when the puzzle difficulty is increased, the system load for the server is decreased. From the two figures, one can observe that the work is indeed shifted from the server to the client, which is one of the intended effects of pTCP.

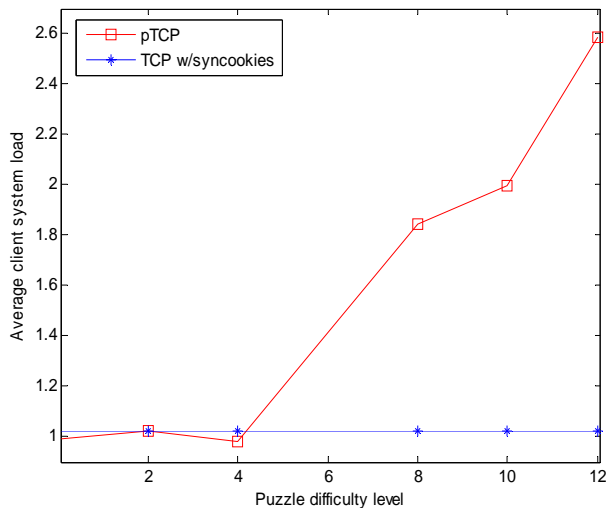


Fig. 10a. Average system load for client during flash-crowd.

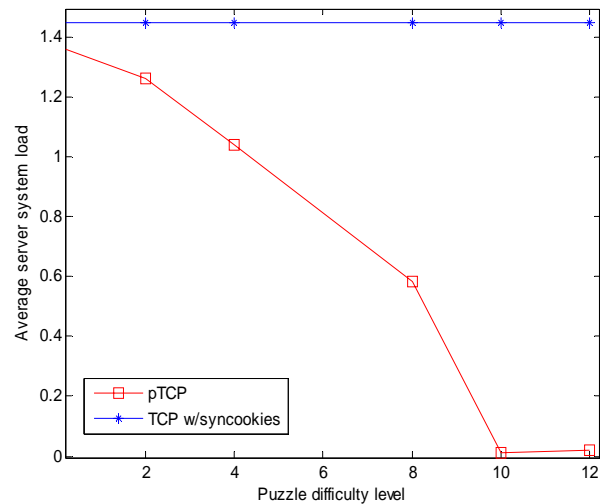


Fig. 10b. Average system load for server during flash-crowd.

## 6. FUTURE WORK AND FURTHER DISCUSSIONS

### 6.1. FUTURE WORK WITH pTCP

There are many future avenues of research in this topic. A client puzzle protocol can be an effective defense for many forms of attacks. As we mentioned earlier, if the goal of the attacker is to actually complete the connection, syncache and syncookies are not valid lines of defense. A client puzzle protocol can also be effective in traffic management and defending against SPAM mail. The concept of reducing SPAM mail by using a cryptographic puzzle is discussed in [30]. By placing a computation problem within the TCP layer, the amount of SPAM mail would most likely be reduced. SPAM can be considered an application layer based attack because it is considered a nuisance and can degrade service for other clients. In our future work, we intend to show how pTCP can be used effectively to reduce the amount of SPAM mail.

### 6.2. IMPLEMENTING CLIENT PUZZLES AT THE NETWORK LAYER

In this paper, we have presented a client puzzle protocol built within the transport layer. Despite the effectiveness of the protocol, it is only effective for combating TCP-based resource exhaustion attacks and can mitigate TCP-based application layer attacks. While TCP is heavily used throughout the Internet, there are several large scale DoS/DDoS attack programs that utilize UDP and ICMP flooding. Thus, the most effective client puzzle protocol will need to be implemented at the network layer (IP layer) [18, 23].

Designing a client puzzle protocol at the network layer raises several questions and introduces new problems that need to be solved. Since TCP is a connection-oriented protocol and utilizes a handshaking mechanism in the initial connection phase, it is natural to transform this handshaking mechanism to incorporate a puzzle challenge. Since IP (and UDP) is a connectionless protocol, the architecture and design of a client puzzle protocol at the network layer will need to be radically different. We have summarized some of the issues and constraints with an IP puzzle scheme and have listed them below:

- Since there is no handshaking involved, clients should have the ability to solve for puzzles before contacting its remote destination or any intermediate destination.
- Intermediate nodes or routers should be able to quickly verify puzzles and pass the packet on to the next hop without any additional communication with the original client.
- If the puzzle has not been solved correctly, the node or router should drop the packet. Doing so will help combat bandwidth consumption DDoS attacks. It is less likely that the bandwidth near a victim will be fully consumed if malicious packets are dropped earlier or further upstream.
- The puzzle challenge mechanism should not require a puzzle to be solved for every packet. To avoid this, it may be possible to challenge a client for the first packet in the flow [18]. A puzzle challenge for every individual packet might introduce unacceptable delays for certain applications (i.e. streaming video, audio applications).

Designing a scalable network layer puzzle protocol is a topic of future research. Although the constraints we have listed above make the task of designing such a protocol very difficult, the benefits of such a protocol is obvious—since the network layer (i.e., IP) is shared by all applications, a DoS attack mitigation capability at this layer would provide the most comprehensive defense mechanism.

### 6.3. FAIRNESS AND PUZZLE DIFFICULTY

The solve time for a computational puzzle will always depend upon the processing power of the computer that is solving the puzzle. In such a scheme, a more powerful client will always have the upper hand over less powerful clients. This drawback is shared by most puzzle schemes, including pTCP. As part of our future work, we plan to explore methods that are able to adjust the level of difficulty for each potential client. By “customizing” the difficulty level for each client, we can prevent more powerful clients (possibly malicious ones) from having an unfair advantage [23]. The difficulty level in pTCP currently is set by the system administrator and can be changed if necessary. The notion of fairness is an important research issue in client puzzles, because more effective protection against DoS attacks can be provided by distributing harder puzzles to clients that demonstrate attacker-like behavior. Puzzles may also become too difficult to solve for weaker legitimate clients and could create a DoS in return, causing weaker clients to spend more time solving a puzzle. Wang and Reiter [12] attempted to address this issue by introducing an auction-based puzzle scheme that allows clients to bid for a connection. However, their scheme also fails to provide fairness—the puzzle solve time is always relative to the computer’s processing power, and thus giving an unfair advantage to bidders with more processing power. This problem may be resolved by using time-based cryptography [31] to create client puzzles. However, at the present time, there is no known time-based cryptosystem that is efficient enough to be used in a client puzzle protocol.

## 7. CONCLUSIONS

This paper has described the architecture of a client puzzle protocol that we call pTCP. This protocol was implemented into the TCP stack in Linux. pTCP has the capability to combat a wide of range of attacks that take advantage of the vulnerabilities of the TCP protocol. Not only can pTCP defend against synflood attacks and help mitigate flash-crowds, as we have shown with our simulation results, it also has the capability to defend against other attacks that may exist at the application layer.

By implementing a puzzle algorithm within the transport layer, we managed to create a client puzzle protocol that is far more effective than puzzle schemes implemented in the application layer. pTCP is an efficient client puzzle protocol that generates and verifies puzzles quickly. The algorithm selected for the client puzzle can be implemented on almost any platform. Since the XTEA6 encryption algorithm is only roughly 4 lines of code, and the puzzle solving algorithm is quite simple, this scheme can be implemented in embedded devices where program space might be an issue. pTCP was also designed for backwards compatibility. Clients operating with pTCP can always connect with servers that use standard TCP. Likewise, clients using standard TCP can connect with servers using pTCP, as long as puzzles are not required to be solved.

The contributions of this paper can be summarized as follows: (i) Presented the architecture, design, and implementation details of a novel client-puzzle protocol embedded within the transport layer, called pTCP; (ii) Demonstrated for the first time the performance advantages of a block-cipher based client puzzle protocol compared to previously proposed hash-based client puzzle protocols; (iii) Demonstrated pTCP’s capability to successfully throttle the client’s connection attempts and reduce the server’s processing load in a flash-crowd scenario.

## References

- [1] E. Skoudis. *Counter Hack. A Step-by-Step Guide to Computer Attacks and Effective Defenses*. Prentice Hall, Upper Saddle River, NJ, 2002.
- [2] J. F. Kurose and K. W. Ross. *Computer Networking. A Top-Down Approach Featuring the Internet, 2<sup>nd</sup> Edition*. Addison Wesley, Boston, MA, 2003.
- [3] C. Jin, H. Wang, and K. G. Shin. "Hop-Count Filtering: An Effective Defense Against Spoofed DDoS Traffic," in *ACM Conference on Computer and Communications Security (CCS'03)*. October 27-31, 2003.
- [4] A. Yaar, A. Perrig, and D. Song. "Pi: A Path Identification Mechanism to Defend against DDoS Attacks," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. May 2003.
- [5] DARPA Internet Program. "RFC 793: Transmission Control Protocol," September 1981.
- [6] CERT@ Advisory CA-1999-17 Denial-of-Service Tools. Available at: <http://www.cert.org/advisories/CA-1999-17.html>. December, 1999.
- [7] Packet Storm Security. Online security reference available at: <http://packetstormsecurity.org/>
- [8] C. Shields. "What do we mean by Network Denial of Service?" in *Proceedings of the 2002 IEEE Workshop on Information Assurance and Security*. United States Military Academy, West Point, NY, 17-19 June 2002.
- [9] J. Lemon. "Resisting SYN flood DoS attacks with a SYN cache", in *Proceedings of USENIX BSDCon 2002*.
- [10] A. Juels and J. Brainard. "Client Puzzles: A cryptographic defense against connection depletion attacks," in *Proceedings of NDSS '99 (Networks and Distributed Systems Security)*, 1999, pages 151-165.
- [11] T. Aura, P. Nikander, and J. Leiwo. "DoS-Resistant Authentication with Client Puzzles," *Lecture Notes in Computer Science*, vol. 2133, 2001.
- [12] X. Wang and M. K. Reiter. "Defending Against Denial-of-Service Attacks with Puzzle Auctions (Extended Abstract)," in *Proceedings of the 2003 IEEE Symposium on Security and Privacy*. 2003.
- [13] B. Bencsath, I. Vajda, and L. Buttyan. "A Game Based Analysis of the Client Puzzle Approach to Defend Against DoS Attacks," in *Proceedings of SoftCOM 2003. International Conference on Software, Telecommunications and Computer Networks*.
- [14] D. Dean and A. Stubblefield. "Using Client Puzzles to Protect TLS," in *Proceedings of the 10<sup>th</sup> USENIX Security Symposium*. August, 2001.
- [15] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. "Moderately Hard, Memory-bound Functions," in *Proceedings of the 10<sup>th</sup> Annual Network and Distributed System Security Symposium*, 2003.
- [16] R. C. Merkle. "Secure Communications Over Insecure Channels," in *Communications of the ACM*. April, 1978.
- [17] B. Waters, J. A. Halderman, A. Juels, and E. W. Felten. "New Client Puzzle Outsourcing Techniques for DoS Resistance." *To Appear in 11th ACM Conference on Computer and Communications Security*. 2004.
- [18] W. Feng, E. Kaiser, W. Feng, and A. Luu. "The Design and Implementation of Network Puzzles," *GI CSE Technical Report CSE-04-003*, August 2004.
- [19] D. Wheeler and R. Needham. "TEA, a Tiny Encryption Algorithm," Unpublished Manuscript. Available at: <http://www.ftp.cl.cam.ac.uk/ftp/papers/djw-rmn/djw-rmn-tea.html>. November, 1994.
- [20] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. "Handbook of Applied Cryptography." CRC Press, 1996.
- [21] J. Kelsey, B. Schneier, and D. Wagner. "Related-Key Cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA," in *1997 International Conference on Information and Communications Security*. Beijing 1997.
- [22] D. Wheeler and R. Needham. "TEA Extensions," Unpublished Manuscript. Available at: <http://www.cl.cam.ac.uk/ftp/users/djw3/xtea.ps>
- [23] W. Feng. "The Case for TCP/IP Puzzles," in *Proceedings of the ACM SIGCOMM 2003 Workshops*. August, 2003.
- [24] J. Crowcroft and I. Phillips. "TCP/IP and Linux Protocol Implementation." John Wiley & Sons, Inc., New York. 2002.
- [25] C. Devine. MD5 Source Code. Available at: <http://www.cr0.net:8040/code/crypto/md5/>.
- [26] D. Ireland, P. Gutmann, and AM Kuchling. SHA-1 Source Code. Available at: <http://www.di-mgt.com.au/crypto.html#sha1>
- [27] I. Ari, B. Hong, E. L. Miller, S. A. Brandt, and D. D. E. Long. "Managing Flash Crowds on the Internet," in *the 11<sup>th</sup> IEEE/ACM International Symposium on Modeling, Analysis, and Simulation of Computer Telecommunications Systems (MASCOTS'03)*. 2003.
- [28] D. Mosberger and T. Jin. "httpperf – A Tool for measuring Web Server Performance," in *First Workshop on Internet Server Performance, ACM*. June, 1998.
- [29] N. J. Gunther. "Linux Load Average." Unpublished Manuscript. Available at: [http://www.luv.asn.au/overheads/NJG\\_LUV\\_2002/luvSlides.html](http://www.luv.asn.au/overheads/NJG_LUV_2002/luvSlides.html)

- [30] C. Dwork and M. Naor. "Pricing via Processing or Combating Junk Mail," *In Advances in Cryptology – Crypto '92*. Springer-Verlag, LNCS volume 740, pp. 129-147, August 1992.
- [31] R. L. Rivest, A. Shamir, and D. Wagner. "Time-lock puzzles and timed-release crypto," Unpublished Manuscript. March, 1996.