A Client-Transparent Approach to Defend Against Denial of Service Attacks

Mudhakar Srivatsa[†], **Arun Iyengar**[‡], **Jian Yin**[‡] **and Ling Liu**[†]
College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA[†]
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA[‡]
{mudhakar, lingliu}@cc.gatech.edu, {aruni, jianyin}@us.ibm.com

Abstract. Denial of Service (DoS) attacks attempt to consume a server's resources (network bandwidth, computing power, main memory, disk bandwidth etc) to near exhaustion so that there are no resources left to handle requests from legitimate clients. An effective solution to defend against DoS attacks is to filter DoS attack requests at the earliest point (say, the web site's firewall), before they consume much of the server's resources. Most defenses against DoS attacks attempt to filter requests from inauthentic clients before they consume much of the server's resources. Client authentication using techniques like IPSec or SSL may often require changes to the client-side software and may additionally require superuser privileges at the client for deployment. Further, using digital signatures (as in SSL) makes verification very expensive, thereby making the verification process itself a viable DoS target for the adversary. In this paper, we propose a light-weight client transparent technique to defend against DoS attacks with two unique features: (i) Our technique can be implemented entirely using JavaScript support provided by a standard client-side browser like Mozilla FireFox or Microsoft Internet Explorer. Client transparency follows from the fact that: (i) no changes to client-side software are required, (ii) no client-side superuser privileges are required, and (iii) clients (human beings or automated clients) can browse a DoS protected website in the same manner that they browse other websites. (ii) Although we operate using the client-side browser (HTTP layer), our technique enables fast IP level packet filtering at the server's firewall and requires no changes to the application(s) hosted by the web server. In this paper we present a detailed design of our technique along with a detailed security analysis. We also describe a concrete implementation of our proposal on the Linux kernel and present an evaluation using two applications: bandwidth intensive Apache HTTPD and database intensive TPCW. Our experiments show that our approach incurs a low performance overhead and is resilient to DoS attacks.

Keywords. Authentication, Availability, Client Transparency, Denial of Service (DoS) attacks, Web Servers

1 Introduction

Recently we have seen increasing numbers of denial of service (DoS) attacks against online services and web applications either for extortion reasons, or for impairing and even disabling the competition [6, 21, 25]. DoS attacks attempt to consume a server's resources (network bandwidth, computing power, main memory, disk bandwidth etc) to near exhaustion so that there are no resources left to handle requests from legitimate clients. These DoS attacks are increasingly mounted by professional attackers using huge zombie nets consisting of thousands of compromised machines on the Internet [16, 30, 27]. An FBI affidavit [6] describes a DoS attack on an e-Commerce website using a 5,000 node zombie net that caused a loss of several millions of dollars in revenue.

Countering DoS attacks on web servers has become a very challenging problem. An effective solution to defend against DoS attacks is to filter attack requests at the earliest point (say, the web site's firewall), before it consumes much of the server's resources. The attack requests arrive from a large number of geographically distributed machines, and thus cannot be filtered on their source IP prefix. Some websites require password and login information before a client can access the website. However, checking the site-specific password requires establishing a connection and allowing unauthenticated clients to access socket buffers and worker processes, making it easy to mount an attack on the authentication mechanism itself. Some sites use strong digital signature based transport level authentication mechanisms [8]; however, the complex server-side computations required for verifying a digital certificate allow an adversary to target the handshake protocol for launching DoS attacks [23].

In general, authentication mechanisms that operate at a higher layer in the networking stack allow an attacker to target lower layers. Some websites may use message authentication codes (MAC) based IP level authentication mechanisms like IPSec [20]. IPSec with preconfigured keys allows packets from unauthenticated clients to be dropped by the firewall. Hence, unauthenticated clients cannot access even low level server resources like socket buffers and transmission control blocks (TCBs). However, IPSec breaks client transparency in several ways: (i) Installing IPSec requires changes to the client-side networking stack, (ii) Installing IPSec requires superuser privileges at the client, (iii) IPSec permits both manual key set up and key exchange using the Internet key exchange protocol (IKE) [15]. The IKE protocol uses the Diffie-Hellman key exchange protocol. Similar to digital signatures this protocol imposes heavy computational load on the server, thereby allowing an adversary to target the IKE protocol for launching DoS attacks. (iv) Manually setting up shared keys circumvents the expensive IKE protocol. However, manual IPSec key set up requires superuser

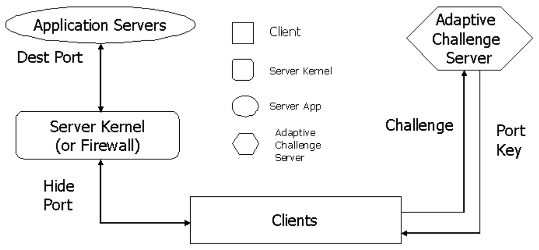


Figure 1: DoS Protection Architecture

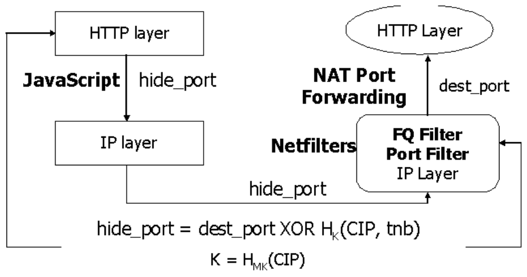


Figure 2: Port Hiding Architecture

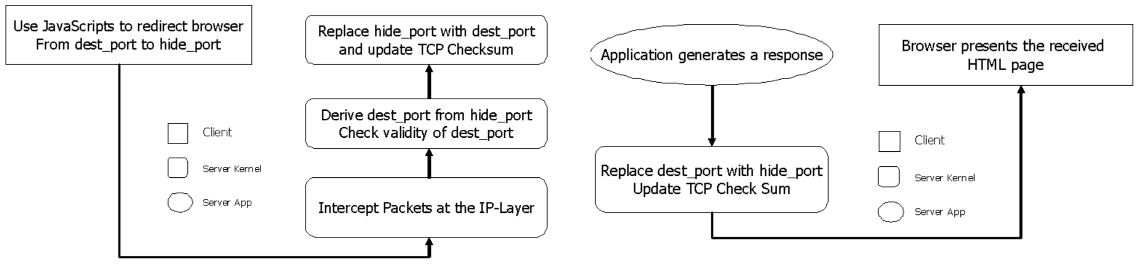

Use JavaScripts to redirect browser From dest_port to hide_port
Replace hide_port with dest_port and update TCP Checksum
Derive dest_port from hide_port Check validity of dest_port
Intercept Packets at the IP-Layer
Client
Server Kernel
Server App
Application generates a response
Replace dest_port with hide_port Update TCP Check Sum
Browser presents the received HTML page
Client
Server Kernel
Server App

Figure 3: Port Hiding Control Flow

privileges at the client.

We observe an inherent conflict between using client authentication for defending against DoS attacks and client transparency. It appears that an effective defense against DoS attacks must operate at lower layers in the networking stack so that the firewall can filter a packet before it can consume much resources at the server. On the other hand, introducing an authentication header at lower layers in the networking stack annuls client transparency, usually by requiring changes to the client-side network stack and by requiring superuser privileges at the client.

Our Approach. In this paper we exploit client-side computations made possible by JavaScripts to embed a weak authentication code into the TCP/IP layer of the networking stack in a client transparent manner. Unlike most authentication protocols that operate between peer layers in the networking stack, our protocol is asymmetric: it operates between the HTTP layer on the client and the IP layer on the server. HTTP level operation at the client permits our implementation to be client transparent, while IP level operation at the server allows packets to be filtered at the server's firewall.

In particular, we embed a 16-bit authenticator in the port number field of TCP packets. This is accomplished at the client's HTTP layer (web browser) using client transparent techniques such as JavaScripts [22]. The server filters IP packets at the firewall based on the authentication code embedded in the destination port field of the packet. If the authentication code were valid, the server uses network address translation (NAT) port forwarding to forward the packet to the actual destination port (say, port 80 for HTTPD). Hence, an unmodified server-side application can seamlessly operate on our DoS protected web server. Although a 16-bit authentication code may not provide strong client authentication, we show that using weak authenticators can significantly alleviate the damage caused by a DoS attack. Our protocol is designed in a way that permits the web server to control the rate at which an attacker can guess the authentication code. This ensures that the web server can tolerate DoS attacks from several thousands of unauthorized malicious clients.

Our Contributions.

IP-layer Filtering. We embed a 16-bit authentication code in the destination port field of a TCP packet. Hence, a firewall at the edge of the server's network can filter unauthenticated IP packets before they enter the network. Filtering packets at the firewall saves a lot of computing, networking, memory and disk resources which would otherwise been expended on processing the packet as it traverses up the server's networking stack.

Client Transparency. Our proposed solution is client transparent, that is, a human being or an automated client-side script can browse a DoS protected website in the same way it browsed an unprotected website. Our DoS protection mechanism does not require any changes to the client-side software or require superuser privileges at the client. All instrumentation required for implementing our proposal can be incorporated on the server side, thereby making our proposal easily deployable.

Application Server Transparency. Our proposed solution is transparent to TCP and higher layers at the web server. Our instrumentation only modifies the IP layer at the firewall to incorporate the authentication check on the destination port field. This permits the web server to layer password and login or digital signatures based authentication protocols atop of our mechanism to achieve strong authentication and resilience to DoS attacks. The applications hosted by the web server can be oblivious to our DoS protection mechanism.

Security Analysis. Using weak authenticators makes it hard but not infeasible for an adversary to guess an authentication code. We present a qualitative analysis of our proposal and present several enhancements to improve its resilience against DoS attacks. Our enhancements largely limit the rate at which the adversary can guess a valid authentication code, thereby making the web server resilient to a DoS attack from several thousands of malicious clients.

Implementation. We describe a detailed implementation of our proposed solution on the Linux kernel and a concrete evaluation using two applications: bandwidth intensive Apache HTTPD benchmark [3] and database intensive TPCW benchmark [31] (running on Apache Tomcat [2] and IBM DB2 [17]). Our experiments show that our proposed solution incurs a low performance overhead and is resilient to DoS attacks.

The remaining sections of this paper are organized as follows. We present our threat model in Section 2 followed by a detailed description of our proposed solution along with a client transparent implementation of our proposal on the Linux kernel in Sections 3 and 4. We describe a detailed evaluation (including performance overhead and resilience to DoS attacks) of our technique in Section 5. We discuss related work in Section 6 and finally conclude in Section 7.

2 Threat Model

In this section, we present a detailed threat model on the web server and the clients. We assume that the web server(s) is honest. All web pages served by a web server are assumed to be valid and correct. However, one could build a feedback mechanism wherein the clients rate the service providers periodically [36]. Over a period of time, clients would access only high quality service providers and the low quality service providers would eventually run out of business.

We assume that the clients may be dishonest. A dishonest client could reveal its authorization information to other unauthorized clients. We assume that the adversary *controls* a set of IP addresses. If an IP address is controlled by an adversary, then the adversary can receive packets sent to that address. On the other hand, if an IP address is not controlled by an adversary, then the adversary can neither observe nor modify packets sent to that address. Nonetheless, the adversary can always spoof the source IP address on a packet with any IP address that is not essentially controlled by the adversary. We assume a bounded strength adversary model. We assume that the adversary has a large but bounded amount of resources at its disposal and thus cannot inject arbitrarily large numbers of packets into the IP network. We assume that the adversary can coordinate activities perfectly to take maximum advantage of its resources; for example, all the compromised zombie computers appearing like a single large computer to the system.

Finally, we do not consider application layer DoS attacks in this paper. For instance an application layer DoS attack may exploit a bug in the application running on the web server to launch DoS attacks. In this paper, we focus on DoS and DDoS attacks that aim at depleting computing and networking resources available at the web server.

3 Port Hiding

3.1 Overview

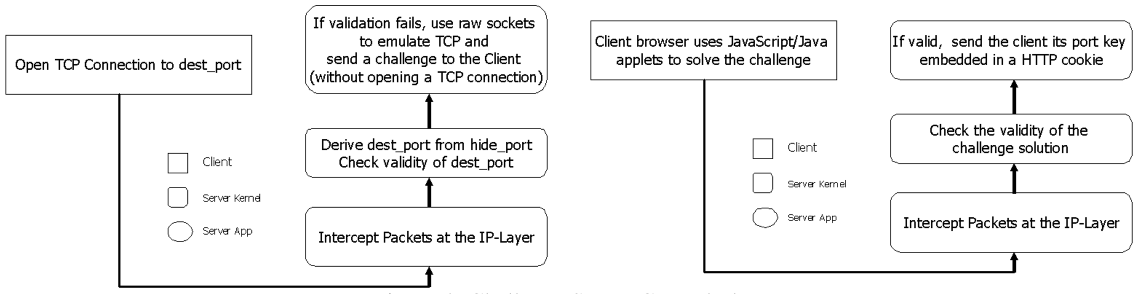
Figure 1 shows a high level architecture of our proposed solution. We achieve resilience to DoS attacks using admission control. Admission control primarily limits the number of concurrent clients accessing the web server. We implement admission control using two components: challenge server and the server firewall (see Figure 1). The challenge server limits the number of active *port keys* issued to clients. A client can efficiently generate a correct authentication code only if it knows its port key. The client-side browser then embeds the authentication code in all TCP packets' destination port field using JavaScript.

We use the server-side firewall to filter IP packets from unadmitted clients. The packet filter at the server drops all non-TCP traffic. Further, it drops all TCP packets that do not have the correct destination port number. Hence, most of the packets sent by clients that do not know their port key would be dropped by the firewall since the authentication check on the packet's destination port number fails. Filtering packets at the firewall significantly reduces the amount of processing, memory, network, and disk resources expended on it. Processing power is saved because the packet does not have to traverse the server's networking stack; additionally, sending an illegal packet to the application layer involves an expensive kernel-to-user context switch (typically, the application runs as a regular user). Memory is saved because the dropped packet does not have to be allocated any space in the memory. Further, if the packet were a TCP ACK packet from an inauthentic client then the web server neither opens a TCP connection nor allocates TCBs. If the incoming packet is from an inauthentic client, network bandwidth is saved because the web server neither receives nor responds to the packet. Finally, by not storing illegal packets in the main memory, the web server may not have to swap pages in/out of the main memory and the hard disk.

A client browsing a DoS protected website has to be capable of: (i) interacting with the challenge server, and (ii) embedding an authentication code in the TCP packet's destination port number field. We achieve both these functionalities in a client transparent manner using JavaScripts at the HTTP layer (web browser). Although we operate at the HTTP layer on the client side, our protocol allows IP layer packet filtering on the server-side firewall. All our instrumentation is done at the server side thereby making the deployment very easy. The instrumentation at the server side includes the challenge server and the firewall.

Challenge Server. The challenge server is used to bootstrap our system by delivering the port keys to admitted clients. A port key is used by a client to compute the authentication code that would be embedded in the destination port number field of the TCP packets. Note that the challenge server itself cannot be protected against DoS attackers using port hiding. Hence, we use a cryptographic challenge [32, 29] based defense mechanism to protect the challenge server. We make the challenge server client transparent by providing a JavaScript to the client that is capable of solving the cryptographic challenge. When the client solves a cryptographic challenge correctly and if the system is capable of handling more clients, then the challenge server would provide the client with a port key. We ensure that solving a cryptographic challenge is several orders of magnitude costlier than the cost of generating the same.

Server Firewall. The server-side firewall is modified to per-



Open TCP Connection to dest_port
If validation fails, use raw sockets to emulate TCP and send a challenge to the Client (without opening a TCP connection)
Derive dest_port from hide_port Check validity of dest_port
Intercept Packets at the IP-Layer
Client
Server Kernel
Server App
Client browser uses JavaScript/Java applets to solve the challenge
If valid, send the client its port key embedded in a HTTP cookie
Check the validity of the challenge solution
Intercept Packets at the IP-Layer

Figure 4: Challenge Server Control Flow

notation	description	default value
N	authentication code size in bits	16 bits
nb	time interval between change in authentication code is 2^{nb} seconds	6
G	number of good clients	100
A	number of bad clients	10^4
r	number of ACK packets per second	1

Table 1: Notation

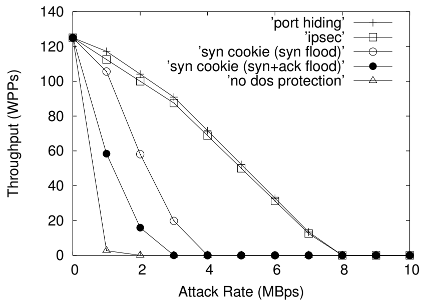


Figure 5: DoS Attack on HTTPD

form two operations on incoming TCP/IP packets: (i) filter packets based on the target port number, and (ii) use probabilistic fair queuing [28] to limit the rate at which packets are sent by a client. The firewall is additionally instrumented to perform port forwarding: the firewall modifies the destination port number on legitimate incoming packets to the application's real port number. Hence, the application layer on the server side can be agnostic to the underlying DoS protection mechanism.

In Section 3.2 we describe a detailed design of our port hiding architecture. We present a client transparent implementation for port hiding in Section 4 and the challenge server. We enhance our technique to handle client-side proxies in Section 6.1. We present a detailed qualitative analysis and several refinements on our proposal in Section 3.3.

3.2 Basic Design

In this section, we present a detailed design for port hiding. Figure 2 shows our architecture for port hiding and Figure 3 shows the operational control flow for port hiding.

The actual destination port ($dest\_port$) is transformed to an authenticated port ($hide\_port$) using a keyed pseudo-random function (PRF) H of the server IP address (SIP), and current time (t) (t is measured as the number of seconds that have elapsed since 1^{st} Jan 1970 GMT) using the port key K as: $hide\_port = dest\_port \oplus H_K(SIP, t_{nb})$. To account for loose time synchronization between the client and the server, we use $t_{nb} = t >> nb$ (instead of t); this allows a maximum clock drift of 2^{nb} seconds. We describe our techniques to handle initial clock skew and clock drifts between the client and server in Section 4.

Observe that the authentication code (and thus $hide\_port$) changes every t_{nb} seconds. Hence, any adversary attempting to guess the authentication code has t_{nb} seconds to do so. At the end of this time interval, the authentication code changes. Even if an adversary guesses the authentication code for one time period it has no useful information about the authentication code for its next period. We further make the authentication code *non-transferable* by ensuring that no two clients get the same port key. We construct a client specific port key as $K = H_{SK(t)}(CIP)$, where the key K is derived from the client's IP-address (CIP) using a PRF H and a time varying server key $SK(t)$. The time varying key $SK(t)$ is derived from the server's master key MK as $SK(t) = H_{MK}(\frac{t}{T})$, where T is the time period for which a port key is valid. The master key MK is maintained a secret by the web server. The admitted clients who possess the key K can send packets to the appropriate destination TCP port. Note that if $hide\_port$ were incorrectly computed, then the reconstructed $dest\_port' = hide\_port \oplus H_K(SIP, t_{nb})$ at the server would be some random port number between (0, $2^{16} - 1$). Hence, one can filter out illegitimate packets using standard firewall rules based on the reconstructed destination port number $dest\_port'$ (say using IP-Tables [1]).

Note that port hiding only prevents unadmitted clients from accessing the service. However, an admitted client may attempt to use a disproportionate amount of resources at the server. We use fair queuing [28] techniques to ensure that an admitted client would not be able to consume disproportionate amount of resources at the server. We add a fair queuing filter immediately above the port hiding filter, that is, all packets coming to this filter had the correct port number on it. Fair queuing ensures that as long as the client's packet arrival rate is smaller than the permissible or the fair packet rate, the probability of dropping the client's packet is zero. Hence, only packets from clients who attempt to use more than their fair share are dropped. It is particularly important not to drop traffic from honest clients, because honest clients use TCP and dropped packets may cause TCP to decrease its window size and consequently affect its throughput. On the other hand, an adversary may be masquerading TCP packets (say, using raw sockets); hence, a dropped packet would not affect an adver-

workload	no DoS protection	port hiding	IPSec
Mix 1 (WIPs)	4.68	4.64 (0.80%)	4.63 (1.0%)
Mix 2 (WIPs)	12.43	12.41 (0.16%)	12.41 (0.18%)
Mix 3 (WIPs)	10.04	10.01 (0.30%)	10.00 (0.37%)
HTTPD (WPPs)	128.00	125.44 (2.06%)	125.40 (2.05%)

Table 2: Port Hiding: Overhead

nb	HTTP (WPPs)	Mix 1 (WIPs)	Mix 2 (WIPs)	Mix 3 (WIPs)
0	94.4 (26.32%)	4.60 (1.80%)	12.33 (0.80%)	9.92 (1.20%)
1	112.32 (12.35%)	4.62 (1.30%)	12.37 (0.48%)	9.96 (0.76%)
2	121.28 (5.37%)	4.63 (1.00%)	12.40 (0.27%)	9.99 (0.52%)
3	125.44 (2.06%)	4.64 (0.80%)	12.41 (0.16%)	10.01 (0.30%)
∞	128.00 (0%)	4.68 (0%)	12.43 (0%)	10.04 (0%)

Table 3: Port Hiding: Varying nb

sary as much it affects an honest client.

3.3 Analysis and Enhancements

In this section, we present a qualitative analysis of our basic port hiding design. We then refine our basic design based on this qualitative analysis to arrive at our final design.

Attacking Weak Authenticators. Since the size of our authentication code is limited to N = 16 bits, a malicious client may be able to discover the destination port corresponding to its IP address with non-trivial probability. Assuming an ideal pseudo-random function (PRF) H, all possible N-bit integers appear to be a candidate $hide\_port$ for a malicious client. For any non-interactive adversarial algorithm, it is computationally infeasible to guess a correct $hide\_port$ with probability greater than 2^{-N}.

Hence, a malicious client is forced to use an interactive adversarial algorithm to guess the value of $hide\_port$. The malicious client may choose a random N-bit integer $rand\_port$ as the destination port number. The client can construct a TCP packet with destination port $rand\_port$ and send the packet to the web server. If the client has some means of knowing that the packet is accepted by the filter, then the client has a valid $hide\_port$ = $rand\_port$. One should note that even if a malicious client successfully guesses the value of $hide\_port$, that value of $hide\_port$ is valid only for the current time epoch. At the end of the time epoch, the malicious client has to try afresh to guess the new value of $hide\_port$. Also observe that using the valid $hide\_port$ value for one epoch does not give any advantage to a malicious client that attempts to guess the $hide\_port$ value for the next epoch.

A Practical Attack. Assuming that the client cannot directly observe the server, the only way for the client to know whether or not the packet was accepted by the firewall is to hope for the web server to respond to its packet. Sending a random TCP packet does not help since the web server's TCP layer would drop the packet in the absence of an active connection. Hence, the malicious client has to send TCP SYN packets with its guess for $hide\_port$. If the web server responds with a TCP SYN-ACK packet then the client has a valid $hide\_port$.

Port Hiding Refinement I. Note that since all N-bit integers appear equally likely to the valid $hide\_port$, the malicious client does not have any intelligent strategy to enumerate the port number space, other than choosing some random enumeration. Clearly, a randomly chosen $hide\_port$ has a 1 in 2^N chance in succeeding thereby reducing the adversarial strength by a huge order of magnitude. Cryptographically, a probability of one in 65,536 (N = 16) is not considered trivially small; however, our techniques can control the rate at which an adversary can break into our system. Observe that the only way a malicious client can possibly infer a valid $hide\_port$ is by probing the server with multiple SYN packets and hoping to receive a SYN-ACK packet from the web server. Now the server could flag a client malicious if it received more than a threshold r number of SYN packets per unit time with an incorrect destination port from the client. Note that one can use our fair queuing filter to rate limit the number of SYN packets per client. This ensures that an adversary has limited to r guesses for $hide\_port$ per second. Hence, the total number of guesses in 2^{nb} seconds is $2^{nb} * r$. Therefore, the probability that a malicious client could guess its authentication code is $min(1, \frac{2^{nb} * r}{2^N}$. Clearly, using a short time interval between changes to authentication code (nb), a smaller permissible rate for SYN packets (r), and a larger authentication code (N) lowers the probability that a malicious client guesses its authentication code.

Attack on Refinement I. However, the technique described above suffers from a drawback. Let us suppose that a malicious client knew the IP address of some legitimate client C. The malicious client could flood the web server with more than r SYN packets per unit time (with randomly chosen destination port numbers) with the packet's source IP address spoofed as CIP, where CIP is the IP address of client C. Now, the firewall would flag the client with IP address CIP as malicious. Hence, all packets sent from the legitimate client C in the future could be dropped by the firewall.

Port Hiding Refinement II. One can circumvent the problem described above using SYN cookies [5] as follows. The web server now responds to all SYN packets (irrespective of whether or not they match the destination port number) with a SYN-ACK packet. The web server encodes a cryptographically verifiable cookie in the TCP sequence number field. When the client sends a TCP ACK packet, the server verifies the cookie embedded in the TCP sequence number field before opening a TCP connection to the client. In addition, the firewall checks the destination port number for all packets except the TCP SYN packet. If a malicious client were to spoof its source IP address in the TCP SYN packet then it would not be able to send a TCP ACK packet with the matching cookie (sequence number) if the IP address CIP is not controlled by the adversary. Recall that our threat model assumes that an adversary would not be able to observe or corrupt any packets sent to an IP address that is not controlled by the adversary. Hence, using SYN cookies eliminates all ACK packets that contain a spoofed source address that is not controlled by the adversary. Now the web server instead of limiting the number of SYN packets per unit time would limit the number of ACK packets per unit time to r. Clearly, the modified technique ensures that an adversary cannot coerce the firewall into dropping packets sent from a legitimate client C.

The rate parameter r cannot be set arbitrarily small in order to accommodate packet losses on the IP network. Typical time out for retransmitting a SYN and ACK packet is about 3000ms. Setting $r = \frac{1}{3}$ ACK packets per second would ensure that a retransmitted packet from a legitimate client would not be dropped by our DoS filter. In our implementation we use a more conservative setting $r = 1$.

4 Client Transparent Implementation

In this section, we present a sketch of our implementation of port hiding. Our implementation operates on both the client and the server. The client-side implementation uses common functionality built into most web browsers and thus does not require any additional software installation. The server-side implementation consists of a loadable kernel module that modifies the IP layer packet processing in the Linux kernel.

Client Side. Port hiding on the client side is implemented entirely using standard JavaScript support available in standard web browsers and does not require any changes to the underlying kernel. In fact, it appears that the destination port number is the only field in the underlying TCP packet that can be directly manipulated using the web browser. We use simple JavaScripts to redirect a request to `protocol://domain:`*hide_port*`/path_name` instead of `protocol://domain/path_name` (usually port numbers are implicit given the protocol: for example, HTTP uses port 80). The port key is made available to the JavaScript by storing it as a standard HTTP cookie on the client browser. We compute $hide\_port$ from $dest\_port$ on the client side using a JavaScript method for MAC (message authentication code) computation. Later in this section, we present techniques to handle the initial clock skew and clock drifts between the client and server. Using JavaScripts makes this approach independent of the underlying OS; also, JavaScripts are available as a part of most web browsers (Microsoft Internet Explorer, Mozilla FireFox).

Server Side. The server-side implementation of port hiding works as follows. The port hiding filter at the server operates at the IP layer in the kernel. The server-side filter uses (Network Address Translation) NAT port forwarding to forward the request from $hide\_port$ to the $dest\_port$. Note that the client-side TCP layer believes that it is connected to $hide\_port$ on the server. Hence, in all server responses we replace the source port from $dest\_port$ to $hide\_port$ so as to make the client believe that the packets are emerging from $hide\_port$. We also appropriately change the TCP checksum when we change the packet's source or destination port. Note that updating the TCP checksum does not require us to scan the entire packet. We can compute the new checksum using the old checksum, $dest\_port$ and $hide\_port$ using simple 16-bit integer arithmetic [9]. We implement these IP-layer filters using NetFilters [1], a framework inside the Linux kernel that enables packet filtering, network address translation and other packet mangling operations.

Additionally, we need every web page served by the server to include a call to the JavaScript that implements port hiding at the client side. One option would be to change all static web pages and the scripts that generate dynamic web pages to embed calls to the port hiding JavaScript. However, we believe that such an implementation would not be feasible. We dynamically modify the HTTP response from the web server to insert calls to JavaScripts using server-side include (SSI [4]). SSI permits us to efficiently inject small additions (function calls to port hiding JavaScript) to the actual HTTP response generated by the web server.

Time Synchronization. We tolerate clock skew and clock drift between clients and the server as follows. First, when the client contacts the challenge server to get the port key, we compute the initial time difference between the client's clock and the server's clock. We include this initial clock skew as a cookie in the HTTP response that includes the client's port key. The client-side JavaScript that updates the authentication code periodically uses the initial clock skew to synchronize the client's local time with that of the server. Assuming that clock drifts are negligibly small, accounting for the initial clock skew is sufficient.

One can additionally tolerate small amounts of clock drifts as follows. The server can update the clock skew cookie each time it sends a HTTP response to the client. Assuming that the clock drift between the client and server does not grow significantly between two successive HTTP responses from the client, an authentic client would be able to compute the correct authentication code. However if the client's think time between successive HTTP requests is very large, then it might be possible that the client's clock drifts more than the permissible level. Even in that case, a client sending IP packets with incorrect authentication headers (destination port number) would be automatically redirected to the challenge server. On solving the challenge, the challenge server would update the cookie that contains the clock skew between the client and the server.

5 Evaluation

In this section, we present three sets of experiments on our proposal. The first set of experiments studies the performance overhead due to the port filter and the fair queue filter. The second set of experiments studies the effectiveness of port hiding against DoS attacks. The third set of experiments studies several attacks on our port hiding filter. All our experiments have been performed on a 1.7GHz Intel Pentium 4 processor running Debian Linux 3.0. We used two types of application servers in our experiments. The first service is a bandwidth intensive Apache HTTPD service [3] running on the standard HTTP port 80. The HTTPD server was used to serve 10K randomly generated static web pages each of size 4 KB. The client-side software was a regular web browser from Mozilla FireFox [11] running on Linux. The web browser was instrumented to programmatically send requests to the server using JavaScripts [22]. The client-side load generator measures the number of web pages downloaded per second (WPPs) as the performance metric. We have also conducted experiments using Microsoft Internet Explorer running on Microsoft Windows XP. The results obtained were qualitatively similar to that obtained using FireFox on Linux and amply demonstrates the portability of our approach and its compatibility across multiple platforms.

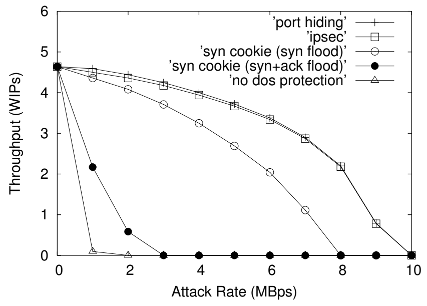


Figure 6: DoS Attack on TPCW

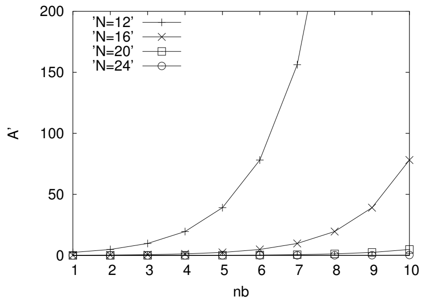


Figure 7: Guessing the Authentication Code

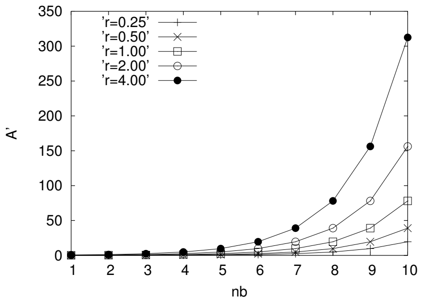


Figure 8: Guessing the Authentication Code

The second service is a database intensive web transaction processing benchmark TPCW 1.0 [31]. We used a Java based workload generator from PHARM [24]. We modified the workload generator to handle HTTP cookies. We used Apache Tomcat 5.5 [2] as our web server and IBM DB2 8.1 [17] as the DBMS. We performed three experiments using TPCW. Each of these experiments included a 100 second ramp-up time, 1,000 seconds of execution, and 100 seconds of ramp-down time. There were 144,000 customers, 10,000 items in the database, 30 entity beans (EBs) and the think time was set to zero (to generate maximum load). The three experiments correspond to three workload mixes built into the client load generator: the browsing mix, the shopping mix and the ordering mix. The TPCW workload generator outputs the number of website interactions per second (WIPs) as the performance metric. In the following portions of this section, we present three sets of experiment on port hiding: (i) measuring its operational overhead, (ii) measuring its resilience to DoS attacks, and (iii) studying attacks on port hiding filter.

5.1 Performance Overhead

Table 2 shows the throughput with and without port hiding using $nb = 3$. Recall from Section 3.2 that 2^{nb} denotes the time interval between changes to the authentication code. The numbers in table 2 are the absolute values and the numbers in brackets show the percentage drop in throughput. The percentage overhead for TPCW is smaller than for HTTP. TPCW being a web transactional processing workload incurs lot of computing and database costs other than simple networking cost. The average traffic between the client load generator and the web server was about 400-600 Kbps; while that for a HTTPD server was about 40-60 Mbps. Table 2 also shows that our approach incurs comparable overhead to that of IPSec. A key advantage of our approach over IPSec is that our approach preserves client transparency.

Table 3 shows the average throughput when we vary nb. Note that with port hiding we vary the $hide\_port$ at time period of 2^{nb} seconds. The numbers in table 3 are the absolute value and the number in brackets show the percentage drop in throughput for different values of nb. We observed that the drop in throughput due to JavaScripts accounted for less than 12% of the total overhead. The biggest overhead in port hiding is due to TCP slow-start [7]. Note that each time the $hide\_port$ changes the client has to open a new TCP connection. Hence, a high bandwidth application like HTTPD suffers from a higher loss in throughput; however, for low bandwidth applications like TPCW the overhead due to TCP slow start is very small.

5.2 Resilience to DoS Attacks

We perform two sets of experiments to study the effectiveness of port hiding in defending against DoS attacks. We describe these experiments below. We have simulated two types of clients: up to 100 good clients and up to 10^4 DoS attackers connected via a 100 Mbps LAN to the server's firewall. The web server is isolated from the LAN connecting the clients; all interactions between the clients and the web server go through the firewall. All the clients compete for the 100 Mbps bandwidth available to the firewall. The good clients were used to measure the throughput of the web server under a DoS attack. The intensity of a DoS attack is characterized by the rate at which attack requests are sent out by the DoS attackers. We measure the performance of the server under the same DoS attack intensity for various DoS filters. Our experiments were run till the *breakdown point*. The breakdown point for a DoS filter is defined as the attack intensity beyond which the throughput of the server (as measured by the good client) drops below 10% of its throughput under no attack. Our experiments show that the breakdown point for the port hiding filter is comparable to that of non client-transparent approaches like IPSec. In fact, we observed that the breakdown for port hiding filters in most cases equals the request rate that almost exhausts all the network bandwidth available to the server. Under such bandwidth exhaustion based DoS attacks, the server needs to use network level DoS protection mechanisms like IP trace back [26, 37] and ingress filtering [10].

Figure 5 and Figure 6 show the effect of DoS attack by malicious clients on the web server. We measured the throughput of the web server as we increase the attack traffic rate. We compare port hiding with other techniques such as IPSec and SYN-cookie. For SYN cookies we measured the effect of performing both SYN flooding and SYN+ACK flooding attack. In a SYN flooding attack the attacker floods the server with many SYN packets. SYN cookies defend the server from SYN flooding attack by embedded authentication information in the TCP sequence number field. In a SYN+ACK flooding attack, the attackers flood the server with SYN packets; wait for the SYN-ACK packet from the server, and respond with an ACK-packet. Hence, the TCP connection is completed at the server, causing the server to construct the state for that connection.

Note that IPSec and port hiding are resilient to SYN+ACK

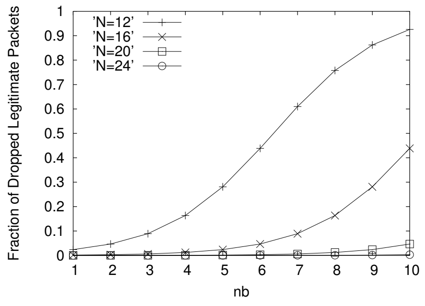


Figure 9: Fraction of Dropped Legitimate Packets

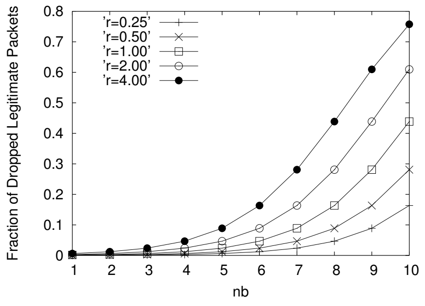


Figure 10: Fraction of Dropped Legitimate Packets

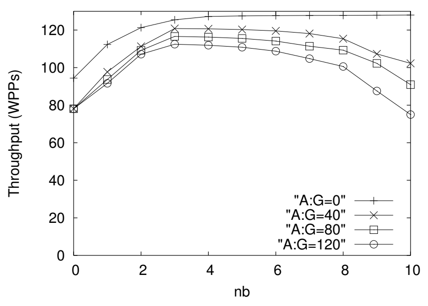


Figure 11: Application Level Throughput under DoS Attacks for HTTPD

floods since an ACK packet with an incorrect authentication code is dropped by the firewall. Observe that the throughput for HTTPD drops almost linearly with the attack traffic rate since HTTPD is a bandwidth intensive application. This is because the incoming attack traffic and the web server response traffic share the 100 Mbps network bandwidth available to the web server. On the other hand, for a less bandwidth intensive application like TPCW, the drop in throughput is very gradual. Observe from the figures that the throughput of the server does not drop to zero unless the adversary soaks up most of the network bandwidth available to the web server. Also the breakdown point for TPCW (9.5 MBps) is much higher than that for HTTPD (7.2 MBps) since TPCW being a database intensive application can operate at high throughput even when most of the network bandwidth is soaked up by the adversary.

Observe that the ability of port hiding and IPSec to defend against DoS attacks is significantly better than SYN cookies. One should also note that IPSec requires changes to the client-side kernel and may require superuser privileges for turning on IPSec and setting up keys at the client. Port hiding on the other hand neither requires changes to the client-side kernel nor requires superuser privileges at the client. This experiment assumes that the adversary uses a randomly chosen authentication code for each IP packet. We study a clever port discovery attack wherein the adversary attempts to guess the $hide\_port$ for bad clients in the next section.

We have also studied the resilience of our challenge server against DoS attacks. The challenge server is largely resilient to TCP layer attacks since we have implemented directly at the IP layer. The challenge server serves three web pages: the challenge page, a JavaScript challenge solver, and the solution verify page directly from the IP layer on the server side. Challenge generation takes 1μs of computing power and generates a 280 Byte web page that contains the challenge parameters. Challenge verification takes 1μs of computing power and generates a 212 Byte web page that sets the port key cookie and the clock skew cookie and redirects the client to the web server (using HTTP redirect). The size of the JavaScript to solve the challenge is about 2 KB. We limit the rate of challenge requests per client to one challenge per second. We limit the download rate for the JavaScript challenge solver per client to one in 64 seconds. This ensures that an attacker cannot throttle the throughput of the challenge server by frequently requesting the JavaScript challenge solver. Note that since the JavaScript challenge solver is a static file and can be cached on the client side for improved performance. Our experiments show the challenge server with 100 Mbps network bandwidth can serve about 44K challenges per second, 59K challenge verifications per second and 6K JavaScript challenge solvers per second.

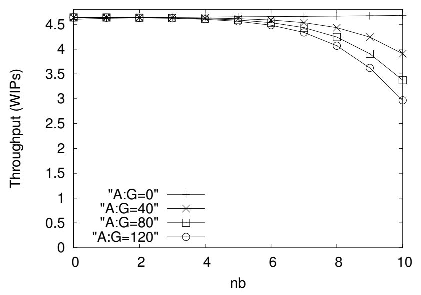


Figure 12: Application Level Throughput under DoS Attacks for TPCW

5.3 Attacks on DoS Filter

As described in Section 3.3, an attack on our DoS filter could proceed in three steps. First, an unauthorized malicious client would attempt to guess its $hide\_port$. Second, those malicious clients that could successfully guess their authentication code could send packets that are accepted by the server-side firewall. Hence, these malicious clients may consume some low level OS resources on the server side including TCP buffers and open TCP connections. This may result in some packets from the legitimate clients being dropped by an overloaded server (at the fair queuing filter). Third, packet loss rate has different effects on the application's overall throughput. For instance, a bandwidth intensive application like HTTPD is affected more due to packet losses (that may result in a TCP window size reduction), as against a database intensive application like TPCW.

Figures 7 and 8 show the number of malicious clients that may potentially guess their authentication codes for different values of nb (time interval between changing authentication codes), N (size of the authentication code), and r (maximum permissible rate for ACK packets per client). Note that the default values of these parameters are specified in Table 1. Even using a weak 16-bit authentication code, the number of attackers who can guess their authentication code (amongst $A = 10^4$ attackers) is very small. This largely restricts the number of unauthorized malicious clients whose packets pass our DoS filter.

Figures 9 and 10 show the fraction of legitimate packets dropped by the fair queuing filter at the firewall for different values of nb, N and r. If we have 10 good clients and A' = 10 bad clients have guessed their authentication code, then the bad clients may potentially use $\frac{10}{10+10}$ = 50% of the server's low level resources such as TCP buffers and open TCP connections. This may consequentially result in some legitimate packets being dropped by the firewall of an overloaded server. Observe that even with several thousand attackers ($A = 10^4$) the fraction of dropped packets is very small.

These dropped packets have different impacts on the application level throughput. Figures 11 and 12 show the application level throughput for HTTPD and TPCW for different values of $A : G$ (ratio of the number of attackers to the number of legitimate clients) and nb. For the HTTPD benchmark the throughput first increases with nb since changing the authentication code infrequently reduces the TCP slow start overhead. However, as nb increases, more attackers may be able to guess their authentication code. This consequently results in dropped packets for legitimate clients resulting in potential reduction in TCP window size (and thus the application level throughput). Even with several thousands (A = 1.2*10^4 in $A : G$ = 120) of attackers, our DoS filter ensures that the throughput of the legitimate clients for both HTTPD and TPCW (as measured under the default settings in Table 1) is about 82% and 94% respectively of its maximum throughput (throughput is maximum when A = 0).

6 Discussion

6.1 Limitations and Open Issues

Client-Side NAT router and HTTP Proxy. In this paper, we have so far assumed that one client IP address corresponds to one client. However, such an assumption may not hold when several clients are multiplexed behind a network address translation (NAT) router or a HTTP proxy. In the absence of a DoS attack there is no impact on the legitimate clients behind a NAT router or a HTTP proxy. However, a DoS attack from a few malicious clients may result in the blockage of all requests from the NAT router's or the HTTP proxy's IP address.

A closer look at the client-side RFC 1631 for the IP NAT [9] shows that client-side NAT routers use port address translation (PAT) to multiplex multiple clients on the same IP address. PAT works by replacing the client's private IP address and original source port number by the NAT router's public IP address and a uniquely identifying source port number. We modify the per client key generation to include the client's IP address and port number as: $K = H_{SK(t)}(CIP, CPN)$, where CIP denotes the IP address of the proxy and CPN refers to the client's translated port number as assigned by the proxy. The client uses key K to derive $hide\_port$ from $dest\_port$.

However, HTTP proxies do not operate using port address translation (PAT). One potential solution is to allow requests only from cooperative HTTP proxies that identify a client using some pseudo identifier. While such a solution retains client anonymity from the web server, it requires cooperation from the HTTP proxies. An efficient proxy transparent solution to handle DoS attacks is an open problem.

Bandwidth Exhaustion Attack. Our approach to client transparent DoS protection protects server-side resources including low level OS resources (TCP buffers, number of open TCP connections) to higher level resources (web server computation & communication, database) from unauthorized malicious clients. However, our approach is vulnerable to bandwidth exhaustion attacks, wherein an adversary throttles all the incoming network bandwidth to the web site using a SYN flooding attack. Wang and Reiter [33] have proposed a technique to mitigate bandwidth exhaustion attacks using congestion puzzles. However, their technique is not client transparent (requires changes to the client-side TCP/IP stack in the kernel). An efficient client transparent solution to mitigate bandwidth exhaustion attacks is an open problem.

6.2 Related Work

One way to defend from DoS attacks is to permit only preauthorized clients to access the web server. Preauthorization can be implemented using TLS/SSL [8] or IPSec [20, 38] with an out of band mechanism to establish a shared key between a preauthorized client and the web server. Now, any packets from a non-preauthorized client can be filtered at the firewall. However, current authorization mechanisms like SSL are implemented on higher layers in the networking stack that permits an attacker to attack lower layers in the networking stack. Further, PKI based authentication mechanisms are computation intensive; this permits an attacker to launch a DoS attack on the authentication engine at the web server. On the other hand, IPSec based authentication is very light weight but requires changes to the client-side kernel and requires superuser privileges at the client. Our proposal simultaneously satisfies client transparency, and yet presents a light weight IP level (weak) authentication mechanism.

Challenge based mechanisms provide an alternative solution for DoS protection without requiring preauthorization. A challenge is an elegant way to throttle the intensity of a DoS attack. For example, an image based challenge (Turing test) [19] may be used to determine whether the client is a real human being or an automated script. Several cryptographic challenges [33, 18, 29, 34] may be used to ensure that the client pays for the service using its computing power. However, most challenge mechanisms make both the good and the bad clients pay for the service, thereby reducing the throughput and introducing inconvenience for the good clients as well. For instance, an image based challenge does not distinguish between a legitimate automated client script and a DoS attack script. In comparison, our proposal is client transparent that neither requires changes to the client-side software nor the presence of a human being on the client side.

Recently, several web applications (including Google Maps [13] and Google Mail [12]) have adopted the Asynchronous JavaScript and XML (AJAX) model [35]. The AJAX model aims at shifting a great deal of computation to the Web surfer's computer, so as to improve the Web page's interactivity, speed, and usability. The AJAX model heavily relies on JavaScripts to perform client-side computations. Similar to the AJAX model we use JavaScripts to perform client-side computations for calculating $hide\_port$ and solving cryptographic challenges. However, our paper focuses on using an AJAX like model to build a client-transparent defense against DoS attacks.

There are several network level DoS protection mechanisms including IP trace back [26], ingress filtering [10], SYN cookies [5] and stateless TCP server [14] to counter bandwidth exhaustion attacks and low level OS resource (number of open TCP connections) utilization attacks. Yang et al.[37] proposes a cryptographic capability based packet marking mechanism to filter out network flows from DoS attackers. These techniques are complementary to our proposal and could be used in conjunction with our solution to enhance the resilience of a web server against DoS attacks.

7 Conclusion

In this paper we have presented a client transparent technique to defend against DoS attacks. We have presented a practical implementation of our proposed solution by embedding an authentication code in the 16-bit destination port number field of a TCP packet. Our proposed solution is client-transparent, application server transparent, and yet permits IP packet level filtering at the web server's firewall. We have also described our implementation using JavaScripts on the client side (HTTP layer) and a pluggable kernel module on the server-side firewall (IP layer). We have qualitatively analyzed our proposal and developed enhancements to achieve higher resilience to DoS attacks by controlling the rate at which an adversary can break the authentication code. A trace based evaluation shows that our technique incurs very low overhead $-$ <12% for a bandwidth intensive HTTPD benchmark and <1% for a non-bandwidth intensive TPCW benchmark. Our evaluation also demonstrates the resilience of our proposal against DoS attacks from 10^4 malicious clients $-$ < 18% drop in throughput for the HTTPD benchmark and <6% drop in throughput for the TPCW benchmark.

References

[1] NetFilter/IPTables project homepage. http://www.netfilter.org/.

[2] Apache. Apache tomcat servlet/JSP container. http://jakarta.apache.org/tomcat, 2004.

[3] Apache. Apache HTTP server. http://httpd.apache.org, 2005.

[4] Apache. Introduction to server side includes. http://httpd.apache.org/docs/howto/ssi.html, 2005.

[5] D. J. Bernstein. SYN cookies. http://cr.yp.to/syncookies.html.

[6] CERT. Incident note IN-2004-01 W32/Novarg.A virus, 2004.

[7] DARPA. RFC 793: Transmission control protocol. http://www.faqs.org/rfcs/rfc793.html, 1981.

[8] T. Dierks and C. Allen. RFC 2246: The TLS protocol. http://www.ietf.org/rfc/rfc2246.txt.

[9] K. Egevang and P. Francis. RFC 1631: The IP network address translator (NAT). http://www.faqs.org/rfcs/rfc1631.html, 1994.

[10] R. Ferguson and D. Senie. RFC 2267: Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. http://www.faqs.org/rfcs/rfc2267.html, 1998.

[11] FireFox. Mozilla firefox web browser. http://www.mozilla.org/products/firefox, 2005.

[12] Google. Google mail. http://mail.google.com/.

[13] Google. Google maps. http://maps.google.com/.

[14] Halfbakery. Stateless TCP/IP server. http://www.halfbakery.com/idea/Stateless_20TCP_2fIP_20server.

[15] D. Harkins and D. Carrel. RFC 2409: The internet key exchange (IKE). http://www.faqs.org/rfcs/rfc2409.html, 1998.

[16] E. Hellweg. When bot nets attack. MIT Technology Review, September 2004.

[17] IBM. DB2 universal database. http://www-306.ibm.com/software/data/db2, 2005.

[18] A. Juels and J. Brainard. Client puzzle: A cryptographic defense against connection depletion attacks. In *NDSS*, 1999.

[19] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds. In *NSDI*, 2005.

[20] S. Kent. Secure architecture for the internet protocol. http://www.ietf.org/rfc/rfc2401.txt, 1998.

[21] J. Leyden. East european gangs in online protection racket. www.theregister.co.uk/2003/11/12/east-european-gangs-in-online/.

[22] Netscape. Javascript language specification. http://wp.netscape.com/eng/javascript/.

[23] B. Pfitzmann and M. Waidner. Attacks on protocols for server-aided RSA computations. In *Eurocrypt*, 1992.

[24] PHARM. Java TPCW implementation distribution. http://www.ece.wisc.edu/ pharm/tpcw.shtml, 2000.

[25] K. Poulsen. FBI busts alleged DDoS mafia. www.securityfocus.com/news/9411, 2004.

[26] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *SIGCOMM*, 2000.

[27] S. Staniford, V. Paxson, and N. Weaver. How to own the internet in your spare time. In *USENIX Security Symposium*, 2002.

[28] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queuing: A scalable architecture to approximate fair bandwidth allocations in high speed networks. In *SIGCOMM*, 1998.

[29] A. Stubblefield and D. Dean. Using client puzzles to protect tls. In *USENIX Security Symposium*, 2001.

[30] L. Taylor. Botnets and botherds. http://sfbay-infragard.org.

[31] TPC. TPCW: Transactional e-commerce benchmark. http://www.tpc.org/tpcw, 2000.

[32] X. Wang and M. K. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *IEEE Symposium on Security and Privacy*, 2003.

[33] X. Wang and M. K. Reiter. Mitigating bandwidth exhaustion attacks using congestion puzzles. In *Proceedings of 11th ACM CCS*, 2004.

[34] B. Waters, A. Juels, A. Halderman, and E. W. Felten. New client puzzle outsourcing techniques for dos resistance. In *ACM CCS*, 2004.

[35] C. K. Wei. AJAX: Asynchronous Java + XML. http://www.developer.com/design/article.php/3526681, 2005.

[36] L. Xiong and L. Liu. Peertrust: Supporting reputation-based trust for peer-to-peer electronic communities. In *Proceedings of IEEE TKDE, Vol. 16, No. 7*, 2004.

[37] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *SIGCOMM*, 2005.

[38] H. Yin and H. Wang. Building an application-aware IPSec policy system. In *USENIX Security Symposium*, 2005.