Mitigating Application Level Denial of Service Attacks on Web Servers: A Client Transparent Approach[1]

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

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. These DoS attacks have increasingly targeted the application level. Application level DoS attacks emulate the same request syntax and network level traffic characteristics as those of legitimate clients, thereby making the attacks much harder to be detected and countered. Moreover, such attacks often target bottleneck resources such as disk bandwidth, database bandwidth, and CPU resources. In this paper we propose to handle DoS attacks using a two-fold mechanism. First, we perform admission control to limit the number of concurrent clients being served by the online service. Admission control is based on port hiding that renders the online service *invisible* to clients who are not authorized, by hiding the port number on which the service accepts incoming requests. Second, we perform congestion control on admitted clients to allocate more resources to good clients. Congestion control is achieved by *adaptively* setting a client's priority level in response to the client's requests in a way that can incorporate application-level semantics. We present a detailed evaluation of the proposed solution using two sample applications: Apache HTTPD and the TPCW benchmark (running on Apache Tomcat and IBM DB2). Our experiments show that the proposed solution incurs low performance overhead and is resilient to DoS attacks.

Categories and Subject Descriptors: C.2.4 [**Distributed Systems**]: Client/Server—*Security and Performance*

Additional Key Words and Phrases: DoS Attacks, Web Servers, Game Theory, Client Transparency

1. INTRODUCTION

Recently, we have seen increasing activities of denial of service (DoS) attacks against online services and web applications to extort, disable or impair the competition. An FBI affidavit [Poulsen 2004] describes a case wherein an e-Commerce website, WeaKnees.com, was subject to an organized DoS attack staged by one of its competitors. These attacks were carried out using sizable 'botnets' (5,000 to 10,000 of zombie machines) at the disposal of the attacker. The attacks began on October 6th 2003, with SYN floods slamming into WeaKnees.com, crippling the site, which sells digital video recorders, for 12 hours straight. In response, WeaKnees.com moved to a more expensive hosting at RackSpace.com. Rackspace.com could condone SYN

[1]Parts of this paper appeared in the Proceedings of IEEE Symposium on Reliable Distributed Systems (SRDS) 2006 [Srivatsa et al. 2006a] and ACM/IFIP/USENIX Middleware Conference 2006 [Srivatsa et al. 2006b]

flooding attacks using SYN-cookies and superior bandwidth capabilities. However, the attackers adapted their attack strategy and replaced simple SYN flooding attacks with a HTTP flood, pulling large image files from WeaKnees.com. At its peak, it is believed that this onslaught kept the company offline for a full two weeks causing a loss of several million dollars in revenue.

As we can see from the example above, sophisticated DoS attacks are increasingly focusing not only on low level network flooding, but also on application level attacks that flood victims with requests that mimic flash crowds [Kandula et al. 2005]. Countering such DoS attacks on web servers requires us to solve two major challenges: application level DoS attacks and client transparency.

1.1 Application Level DoS Attacks

Application level DoS attacks refer to those attacks that exploit application specific semantics and domain knowledge to launch a DoS attack such that it is very hard for any DoS filter to distinguish a sequence of attack requests from a sequence of legitimate requests. Two characteristics make application level DoS attacks particularly damaging. First, application level DoS attacks emulate the same request syntax and network level traffic characteristics as that of the legitimate clients, thereby making them much harder to detect. Second, an attacker can choose to send expensive requests targeting higher layer server resources like sockets, disk bandwidth, database bandwidth and worker processes [CERT 2004][Leyden 2003][Poulsen 2004].

As in the case of WeaKnees.com, an attacker does not have to flood the server with millions of HTTP requests. Instead, the attacker may emulate the network level request traffic characteristics of a legitimate client and yet attack the server by sending hundreds of resource intensive requests that pull out large image files from the server. An attacker may also target dynamic web pages that require expensive search operations on the backend database servers. A cleverly constructed request may force an exhaustive search on a large database, thereby significantly throttling the performance of the database server.

There are two major problems in protecting an online e-Commerce website from application level DoS attacks. First, application level DoS attacks could be very subtle making it very hard for a DoS filter to distinguish between a stream of requests from a DoS attacker and a legitimate client. In section 2 we qualitatively argue that it would be very hard to distinguish DoS attack requests from the legitimate requests even if a DoS filter were to examine any statistics (mean, variance, etc) on the request rate, the contents of the request packet headers (IP, TCP, HTTP, etc) and even the entire content of the request packet itself. Second, the subtle nature of application level DoS attacks make it very hard to exhaustively enumerate all possible attacks that could be launched by an adversary. Hence, there is a need to defend against application level DoS attacks *without knowing* their precise nature of operation. Further, as in the case of WeaKnees.com, the attackers may continuously change their strategy to evade any traditional DoS protection mechanisms.

1.2 Client Transparency

An effective solution to defend against DoS attacks must be client transparent, that is, it should neither require any changes to the client software stack nor require the client to have superuser privileges. Additionally, a client, be it a human user or an automated script, must be able to interact with the Web server without requiring functional changes. However, one observes 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 significant resources at the server. Note that solutions that operate at higher layers on the networking stack are vulnerable to the 'attacks from below'. On the other hand, introducing changes into 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.

For example, let us consider message authentication codes (MAC) based IP level authentication mechanisms like IPSec [Kent 1998] that permit only authorized clients to access the Web server. IPSec with preconfigured keys allows packets from unauthorized clients to be dropped by the firewall. Hence, unauthorized 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) [Harkins and Carrel 1998]. The IKE protocol uses the Diffie-Hellman key exchange protocol. This protocol imposes heavy computational load on the server similar to that imposed by digital signatures, 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 privileges at the client.

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) [Kandula et al. 2005] may be used to determine whether the client is a real human being or an automated script. Several cryptographic challenges [Wang and Reiter 2004][Juels and Brainard 1999][Stubblefield and Dean 2001][Waters et al. 2004] 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, this paper focuses on a client transparent approach that neither changes to the client-side software nor requires the presence of a human being on the client side.

1.3 Our Approach

In this paper we propose a client-transparent mechanism to protect a server from application level DoS attacks. The proposed solution to handle DoS attacks uses a two-fold mechanism. First, we perform admission control to limit the number of concurrent clients being served by the online service. Admission control is based on port hiding that renders the online service 'invisible' to clients who are not admitted (or authorized), by hiding the port number on which the service accepts incoming requests. Second, we perform congestion control on admitted clients to allocate more resources to good clients. Congestion control is achieved by 'adaptively' setting a client's priority level in response to the client's requests, in a way that can incorporate application level semantics.

We exploit client-side computations made possible by JavaScripts to embed a weak authentication code and a client priority level 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 (implemented using standard web browser), while IP level operation at the server allows packets to be filtered at the server's firewall. 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.

In particular, our admission control mechanism embeds 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 [Netscape]. 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 congestion control filter uses a similar client transparent technique to embed a client's priority level into all its HTTP requests. A client's priority level is used to throttle its request rate at the server's firewall (IP-layer filtering). This priority level is itself continuously updated in way that mitigates application level DoS attacks as follows. Unlike traditional DoS protection mechanisms that attempt to distinguish a DoS attack request from the legitimate ones, our congestion control filter examines the amount of resources expended by the server in handling a request. We use the utility of a request and the amount of server resources incurred in handling the request to compute a score for every request. We construct a feedback loop that takes a request's score as its input and updates the client's priority level. In its simplest sense, the priority level might encode the maximum number of requests per unit time that a client can issue. Hence, a high scoring request increases a client's priority level, thereby permitting the client to send a larger number of requests per unit time. On the other hand, a low scoring request decreases a client's priority level, thereby limiting the number of requests that the client may issue per unit time. Therefore, application level DoS attack requests that are resource intensive and have low utility to the e-commerce website would decrease the attacker's priority level. As the attacker's priority level decreases, the intensity of its DoS attack decreases.

An obvious benefit that follows from the description of our congestion control filter is that it is independent of the attack's precise nature of operation. As pointed out earlier it is in general hard to predict, detect, or enumerate all the possible attacks that may be used by an attacker. Also, a mechanism that is independent of the attack type can implicitly handle intelligent attackers that adapt and attack the system. Indeed any adaptation of an application level DoS attack would result in heavy resource consumption at the server without any noticeable changes to the request's syntax or traffic characteristics. Further, our mechanism does not distinguish requests based on the request rate, the packet headers, or the contents of the request. We illustrate in Section 2 that it is very hard to distinguish an attack request from the legitimate ones using either the request rate or the contents of the request.

In this paper we describe a detailed implementation of our DoS protection filters using a JavaScript capable web browser[2] (client) and the Linux kernel (firewall). We present a detailed evaluation of the proposed solution using two sample applications: Apache HTTPD [Apache 2005a] and the TPCW benchmark [TPC 2000] (running on Apache Tomcat [Apache 2004] and IBM DB2 [IBM 2005]). Our experiments show that the proposed solution incurs low performance overhead (about 1-2%) and is resilient to DoS attacks. We demonstrate the drawbacks in traditional DoS filters including, violation of client transparency and vulnerability to application level DoS attacks, and experimentally demonstrate the efficacy of our DoS filters against a wide range of attacks.

2. OVERVIEW

2.1 Application Level DoS Attacks: Examples

Example 1. Consider an e-Commerce website like WeaKnees.com. The HTTP requests that pulled out large image files from WeaKnees.com constituted a simple application level DoS attack. In this case, the attackers (a collection of zombie machines) sent the HTTP requests at the same rate as a legitimate client. Hence, a DoS filter may not be able to detect whether a given request is a DoS attack request by examining the packet's headers, including the IP, the TCP and the HTTP headers. In fact, the rate of attack requests and the attack request's packet headers would be indistinguishable from the requests sent by well behaved clients.

Example 2. One could argue that a DoS filter that examines the HTTP request URL may be able to distinguish DoS attackers that request a large number of image files from that of the good clients. However, the attackers could attack a web application using more subtle techniques. For example, consider an online bookstore application like TPCW [TPC 2000]. As with most online e-Commerce applications, TPCW uses a database server to guarantee persistent operations. Given an HTTP request, the application logic transforms the request into one or more database queries. The cost of a database query not only depends on the type of the query, but also depends on the query arguments. For instance, an

[2]JavaScript is supported by most modern browsers including Microsoft IE and Mozilla FireFox

HTTP request may require an exhaustive search over the entire database or may require a join operation to be performed between two large database tables. This makes it very hard for a DoS filter to detect whether a given request is a DoS attack request by examining the packet's headers and all its contents. In fact, the rate of attack requests and the attack request's packet headers and contents would be indistinguishable from those sent by any well behaved client unless the entire application logic is encoded in the DoS filter. However, this could make the cost of request filtering almost as expensive as the cost of processing the actual application request itself. Indeed a complex DoS filter like this could by itself turn out to be a target for the attackers.

2.2 Threat Model

We assume that an adversary is interested in launching a DoS attack on some web server or application server or any generic web service. An adversary might target its attack on either of the following two entities: the good clients and the web servers. We do not consider DoS attacks directly on the clients in this paper. However, the adversary may target its attack on the web servers.

We assume that the adversary can spoof the source IP address. We also assume that the adversary has a large but bounded number of IP-addresses under its control. If an IP-address is controlled by an adversary, then the adversary can both send and receive packets from that IP-address. We assume that the adversary can neither observe nor modify the traffic to clients whose IP-address are not controlled by the adversary. However, the adversary can always send packets with a spoofed source IP-address that is not essentially controlled by the adversary. We also assume that the adversary has large, but bounded amount of network and computing resources at its disposal and thus, cannot inject arbitrarily large number 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 appears like a single large computer to the system.

We assume that the adversary can perform network layer DoS attacks. Network layer DoS attacks include attacks directed on the TCP/IP layer such as SYN-flooding attack. The network layer attacks aim at exhausting computing, networking and memory sources available at the web server by flooding them with a large number of IP packets or TCP connection requests over a short duration of time.

We assume that the adversary may perform application layer DoS attacks in this paper. Most modern application servers perform complicated operations such as running web services that require heavy weight transactional and database support. Understanding the application semantics may help an adversary to craft a DoS attack by sending a many resource intensive requests. For example, in a typical bookstore application, the adversary may issue frequent search requests that involves expensive database operations. Application layer attacks are hard to be detected at the traditional IP-layer DoS filters primarily because they lack application specific semantics. We discuss some concrete examples of such application level DoS attacks in the next section.

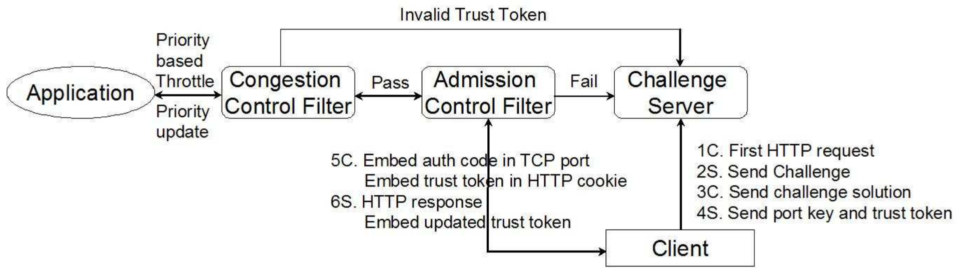


Fig. 1. System Architecture

2.3 System Architecture

We achieve resilience to DoS attacks using two layers of DoS protection: admission control and congestion control. Admission control limits the number of clients concurrently served by the server. The number of permitted concurrent clients can be adaptively varied depending on the server load. Admission control is implemented using destination port number as an authentication code for admitted clients. Without knowing the application's port number it is obviously much harder for unadmitted clients to launch a DoS attack. Most of the unadmitted traffic can be filtered at the IP-layer itself based on its target port number. An unadmitted client would not even be able to launch a TCP SYN-flooding attack. This prevents an unadmitted client packet from consuming computing, network and memory resources at the server as it passes through higher layers in the network stack. In this paper we explore algorithms and present detailed security analyses for using weak (16-bit) authentication codes to defend against DoS attacks.

The congestion filter operates on top of the admission control filter. Congestion control limits the amount of resources allocated to each admitted client. Congestion control is implemented as a trust token that encodes the priority level that a client is eligible to receive. A client's priority level is adaptively varied by the server using application-specific knowledge on the nature of requests made by the client. For instance, one may choose to increase a client's priority level if the client performs a `buy` transaction with the server; and decrease a client's priority level if the client performs resource intensive operations on the server. A client's priority level is used to rate limit the number of requests accepted from the client at the server's firewall. Allowing the application to set a client's priority level permits us to incorporate application specific semantics (domain knowledge) and is thus highly flexible. IP-level (firewall) level filtering for requests ensures that most application level DoS attack requests are dropped before they can extensively consume server-side resources. In this paper, we explore several algorithms to vary the client's priority level and study their effect on the system. We also provide an API for application programmers to define application specific solutions to update a client's priority level.

Our proposed solution is client-transparent in the sense that it neither requires software installation nor human involvement at the client-side. A client (human being or an automated script) can browse a DoS protected web server just as if it browses any other server with standard web browsers (e.g. FireFox, Microsoft IE, etc). All our instrumentation is done at the server side thereby making the deployment very easy. Figure 1 shows our high level architecture including the main components: challenge server, admission control filter, congestion control filter and the application. The figure also shows the order in which a client interacts with these components: messages that originate from the client are labeled C while those from the server are labeled S.

Challenge Server. The challenge server is used to bootstrap our system. The challenge server is responsible for delivering port keys to admitted clients and initializing the client's priority in a trust token. The port key is used by a client to determine the correct target destination port number. The trust token encodes the initial client priority level. Note that the challenge server itself cannot be protected against DoS attackers using port hiding. Hence, we use a cryptographic challenge based defense mechanism to protect the challenge server. 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 the port key. We ensure that solving the cryptographic challenge is several orders of magnitude costlier than generating the port key and initializing the trust token.

Server Firewall. The firewall (IP layer) at the server is modified to perform two operations: (i) filter packets based on the target port number, and (ii) use the client's priority level to filter HTTP requests sent by a client using weighted fair queuing [Stoica et al. 1998].

Application Server. The application layer at the server is modified to perform two operations: (i) implement the client's response time priority, and (ii) use application specific rules to update the throughput priority level of the client. The response time priority is enforced by appropriately setting the request handling thread's priority and network priority (say, using DSCP [Black] or IP-precedence [Nichols et al.]. The client's new throughput priority level may be computed using a utility based model that considers the current request and the set of recent requests sent by the client.

3. DOS PROTECTION MECHANISMS

3.1 Admission Control Filter

We implement admission control using the following steps. (i) First, the client attempts to obtain a *port key* from the challenge server. 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.

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.

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

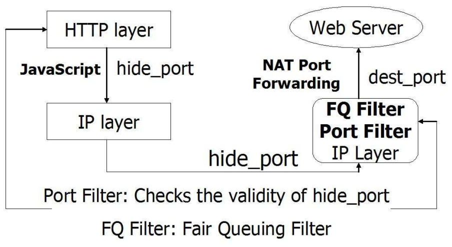


Fig. 2. Port Hiding: Design

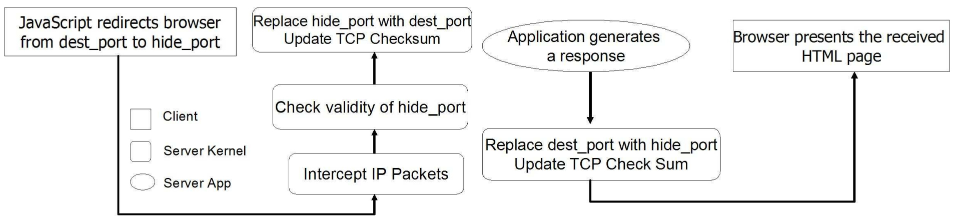

JavaScript redirects browser from dest_port to hide_port
Replace hide_port with dest_port Update TCP Checksum
Application generates a response
Browser presents the received HTML page
Client
Server Kernel
Server App
Check validity of hide_port
Intercept IP Packets
Replace dest_port with hide_port Update TCP Check Sum

Fig. 3. Port Hiding Control Flow: square-with-sharp-corner, square-with-round-corner and oval denote operations carried out by client, server kernel (firewall) and server application respectively

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 in the user domain). 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 (transmission control blocks). 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.

In the following Section we describe a detailed design of our admission control filter. We present a detailed qualitative analysis and several refinements on our proposal in Section 3.1.2. We present a client transparent implementation for port hiding in Section 4.

3.1.1 *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 client IP address (CIP), server IP address (SIP), and current time (t) (t is measured as the number of seconds that have elapsed since 1st Jan 1970 GMT) using the port key K as: $hide\_port = dest\_port \oplus$

$H_K(CIP, SIP, t_{nb})$. 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 $dest\_port' = hide\_port \oplus H_K(CIP, SIP, t_{nb})$ 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 [NetFilter]).

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 [Stoica et al. 1998] techniques to ensure that an admitted client would not be able to consume a disproportionate amount of resources at the server. Fair queuing ensures that as long as the client's packet arrival rate is smaller than 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 adversary as much it affects an honest client.

3.1.2 *Port Hiding: 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.

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 [Bernstein 2005] 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

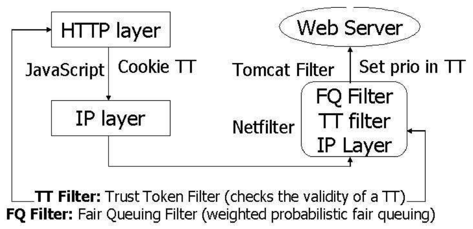

HTTP layer
JavaScript
Cookie TT
IP layer
Web Server
Tomcat Filter
Set prio in TT
Netfilter
FQ Filter
TT filter
IP Layer
TT Filter: Trust Token Filter (checks the validity of a TT)
FQ Filter: Fair Queuing Filter (weighted probabilistic fair queuing)

Fig. 4. Congestion Control Architecture

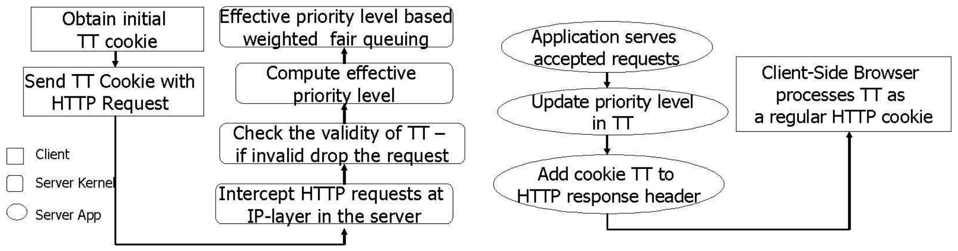

Obtain initial TT cookie
Send TT Cookie with HTTP Request
Client
Server Kernel
Server App
Effective priority level based weighted fair queuing
Compute effective priority level
Check the validity of TT – if invalid drop the request
Intercept HTTP requests at IP-layer in the server
Application serves accepted requests
Update priority level in TT
Add cookie TT to HTTP response header
Client-Side Browser processes TT as a regular HTTP cookie

Fig. 5. Congestion Control: Control Flow

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. Clearly, the modified technique ensures that an adversary cannot coerce the firewall into dropping packets sent from a legitimate client C.

Attack on Refinement II. However, the adversary could still flood the web server with unlimited numbers of SYN packets. The protocol expects the web server to respond with a SYN-ACK packet for all SYN packets irrespective of whether or not the SYN packet matches the authentication code embedded in the destination port number field. One can mitigate this problem by putting an upper bound on the number of SYN packets accepted from a client to at most r SYN packets per unit time. However, as described above, using a finite r permits the adversary to coerce the web server to drop packets from a legitimate client. Clearly, as r increases, it becomes more expensive for an adversary to do so. However, as r increases, the bad clients would be able to flood more SYN packets to the web server. Allowing more SYN packets per unit time permits a bad client to guess its correct $hide\_port$. Hence, the parameter r must be carefully chosen by the web server. We use experimental techniques and analytical results to choose effective DoS defense strategies (see Section 5).

3.2 Congestion Control Filter

The congestion control filter operates on top of the admission control filter. The fundamental idea behind congestion control is to adaptively vary the priority offered to a client depending on the client's behavior in the recent past. For instance, we increase a client's priority level if the client behaves well and decrease it on detecting an application level DoS attack from the client. Our congestion control mechanism is analogous to the TCP congestion control mechanism, with the priority level of a client being analogous to the TCP window size. We use an additive increase and multiplicative decrease style algorithm to manipulate the priority level based on a utility based model of the client's past behavior.

A client's priority level is embedded in a trust token which in turn is sent as a standard HTTP cookie in all responses from the server to the client. Using standard HTTP cookie semantics, a legitimate client would include the trust token in all its future requests to the server. A client presenting a valid trust token to the server would be served at the priority level encoded in the token. Otherwise, the client's request would be served at the lowest priority level. This would motivate the clients to present their trust token and thus obtain better quality of service (for example, better throughput and/or response time) from the server.

The challenge server initializes a client's trust token (along with its port key) when the client first accesses the web server. The trust token includes a message authentication code to ensure that it would be computationally infeasible for a client to undetectably modify the token. We use an IP-level packet filter to filter HTTP requests from admitted clients. The packet filter implements the throughput priority using weighted fair queuing, that is, it ensures that a client with priority level 100 may send twice as many requests (per unit time) than a client with priority level 50. HTTP requests from clients attempting to issue a disproportionately large number of requests (relative to their priority level), are dropped at the IP-layer itself, thereby, significantly reducing the amount of processing / memory / network / disk resources. The application layer implements the response time priority, that is, it ensures that a client with a higher priority level experiences a smaller response time for its requests. The application layer is also responsible for varying the client's throughput priority level using application specific semantics and domain knowledge.

3.2.1 *Trust Tokens.* In this section, we describe how a trust token is constructed. Then, we describe techniques to use the congestion token to defend against application-level DoS attacks.

A trust token (tt) is constructed as follows: $tt = cip$, sip, tv, $priority$, $H_{MK}(cip$, sip, tv, $priority)$, where cip (4 Bytes) denotes the client's IP address, sip (4 Bytes) denotes the server's IP address, tv (4 Bytes) denotes the time at which the trust token was issued (time is expressed as the number of seconds from 1^{st} Jan 1970), $priority$ (2 Bytes) denotes the priority level assigned to the client by the server, MK denotes a secret cryptographic key used by the server and E denotes a symmetric key encryption algorithm (like AES [NIST] or DES [FIPS]). The priority level $priority$ consists of two parts: a one Byte throughput priority level and a one Byte response time priority level. Essentially, this would permit 0-255 possible throughput and response time priority levels for a client.

Figure 4 shows the architecture for congestion control and Figure 5 shows the operational usage of the token. A legitimate client operates as follows. A client obtains its first token tt when it solves a challenge, and the token is stored as an HTTP cookie in the client-side browser. The client includes the token tt in all HTTP requests to the server.

On the server side, we perform two operations. First, we filter and rate control requests to the server at the IP layer in the kernel's network stack (or the firewall). For similar reasons discussed under port hiding, it is important that packets are filtered as soon as possible in order to save computing and memory resources on the server. The server checks if the packet is a HTTP request and if so, it extracts the token tt. It checks if tt is a valid trust token and if so, it extracts the client's priority level. A trust token is valid if the $tt.cip$ matches the client's IP address, $tt.sip$ matches the server's IP address, $tt.tv$ is some time in the past ($tt.tv < currenttime$). If so, the server extracts the priority level from tt; else the request packet is served at the lowest priority level.

An adversary may behave benignly until it attains a high priority level and then begins to misbehave. Consequently, the server would issue a trust token with a lower priority level. However, the adversary may send an old congestion token (with high priority level) to the server in its future requests. We prevent such attacks by computing the effective priority level. The server uses the request's throughput priority level $priority_{thru}$, the time of token issue and the client's request rate to compute the effective priority level $epriority$ as follows: $epriority = priority_{thru} * e^{-\delta * max(t - tt.tv - \frac{1}{r}, 0)}$, where t denotes the current time, $tt.tv$ denotes the time at which the token tt was issued and r denotes the client's request rate and δ denotes a tuneable parameter for computing the penalty incurred when using an old trust token. The key intuition here is that if $t - tt.tv$ is significantly larger than the client's mean inter request arrival time ($\frac{1}{r}$) then the client is probably sending an old congestion token. The larger the difference between $(t - tc.tv)$ and $\frac{1}{r}$, the more likely it is that the client is attempting to send an old token. Hence, the effective priority level $epriority$ drops exponentially as the difference between $(t - tc.tv)$ and $\frac{1}{r}$ increases. Note the fair queuing filter estimates the client request rate r for performing weighted probabilistic fair queuing.

Once the request is accepted by the IP-layer packet filter, the request is forwarded to the application. The application server handles a request based on the request's response time priority level ($priority_{resp}$). The server can improve the response time to clients with higher response time priority levels by setting a higher priority for the threads that handle their requests. In general, the handling thread's priority could be proportional to the request's response time priority level. In three-tiered systems, the application server may need to interact with other application servers or database servers to handle a request. In this case, the servers may use DSCP or IP precedence to decrease the network latencies for requests with higher priority levels.

When the server sends a response to the client, it updates the client's priority level based on several application specific rules and parameters. For example, in an electronic commerce application, if the client were to complete a transaction by providing a valid credit card number, the server could increase the client's priority level. We propose to use a utility based cost function $C(rq) = f(rt, tp, np, ut)$, where rq denotes the client's request, rt is the time taken by the server to generate the response for request rq, tp denotes the thread priority that was used to handle rq, np denotes the network priority used to handle rq, and ut denotes the utility (in dollars for a credit card transaction) of rq. In our first prototype, we use a simple cost function $C(rq) = ut - \gamma * nrt$, where nrt denotes the normalized response time and γ is a tune-able parameter. The normalized response time nrt denotes the effort expended by the server to handle the request rq. nrt is derived as a function of the measured response time rt, the thread priority tp and the network priority np. Note that the higher the value of tp (and np), the lower the response time. Finally, the normalized response time nrt denotes the effort expended by the server; hence it is subtracted from the request's utility ut.

The new priority level could be computed as $priority = g(priority, C)$, where $priority$ is the current effective priority level of the client. In our first prototype, we use TCP style congestion control technique with additive increase and multiplicative decrease as follows: If $C(rq) \geq 0$, then $priority = priority + \alpha * C(rq)$, and $priority$ $= \frac{priority}{\beta * (1 - C(rq)}$ otherwise. The additive increase strategy ensures that the priority level rises slowly as the client behaves benignly; while the multiplicative decrease strategy ensures that the priority level drops quickly on detecting a DoS attack from the client.

In summary, we perform request filtering at the server-side kernel or firewall. As we have pointed out earlier, filtering out bad requests as soon as possible minimizes the amount of server resources expended on them. However, the parameter that determines this filtering process (the client's throughput priority level) is set by the application. This approach permits highly flexible DoS protection, since it is possible to exploit application specific semantics and domain knowledge in computing the client's priority level.

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 IE, Mozilla FireFox).

Server-Side IP-Layer. 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*. 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 [Egevang and Francis 1994]. We implement these IP-layer filters using NetFilters [NetFilter], a framework inside the Linux kernel that enables packet filtering, network address translation and other packet mangling.

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 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 to insert calls to JavaScripts in a web server using server-side include (SSI [Apache 2005b]). SSI permits us to efficiently inject small additions to the actual HTTP response generated by the web server.

Server-Side Application-Layer. We use Apache Tomcat filters to hook on to the processing of a HTTP request. We hook onto an incoming request before the request is forwarded to the servlet engine. This filter is used to record the time at which the request processing started. This filter is also used to implement response time priority embedded in the trust token typically, by setting the request handling thread's priority in proportion to the response time priority level.

Similarly, a filter on an outgoing HTTP response is used to record the time at which the request processing ended. This filter additionally provides an application programmer the following API to use application specific rules and domain knowledge to update the client's priority level after processing a request rq: `priority updatePrio (priority oldPrio, URL requestURLHistory, responseTime rt)`, where `oldPrio` denotes the client's priority level before it issued the request rq, `requestURLHistory` denotes a finite history of requests sent from the client, and `rt` denotes the server response time for request rq. Additionally, this filter encrypts the trust token tt and embeds it as a cookie in the HTTP response.

Sample API Implementations. We now describe three sample implementations of our API to demonstrate its flexibility.

Resource Consumption. In Section 3.2 we presented a technique to update a client's priority level based on its request's response time and utility. Utility of the request can be computed typically from the requesting URL using application specific semantics and domain knowledge; note that client supplied parameters are available as part of the request URL. The response time for a request is automatically measured by our server side instrumentation. Additionally, one could use several profiling techniques to determine low level resource consumption such as CPU cycles, disk bandwidth, etc.

Input Semantics. Many e-commerce applications require inputs from users to follow certain implicit semantics. For example, a field that requests a client's age would expect a value between 1 and 100. One can use the client supplied parameters (that are available as a part of the request URL) to estimate the likelihood that a given request URL is a DoS attack or not. Naive DoS attack scripts that lack complete domain knowledge to construct semantically correct requests (unlike a legitimate automated client side script), may err on input parameter values.

Link Structure. In many web applications and web servers the semantics of the service may require the user to follow a certain link structure. Given that a client has accessed a page P, one can identify a set of possible next pages P_1, P_2, $\cdots$, P_k along with probabilities tp_1, tp_2, $\cdots$, tp_k, where tp_i denotes the probability that a legitimate client accesses page P_i immediately after the client has accessed page P. The server could lower a client's priority level if it observes that the client has significantly deviated from the expected behavior. Note that tracking a client's link structure based behavior requires a finite history of URLs requested by the client.

While heuristics like *Input Semantics* and *Link Structure* can guard the web server from several classes of application level DoS attacks, one should note that these heuristics may not be sufficient to mitigate all application level DoS attacks. For example, a DoS attacker may use requests whose cost is an arbitrarily complex function of the parameters embedded in the request. Nonetheless the *Resource Consumption* based technique provides a solution to this problem by actually measuring the cost of a request, rather than attempting to infer a DoS attack based on the request.

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, a 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 (see Section 4). On solving the challenge, the challenge server would update the cookie that contains the clock skew between the client and the server.

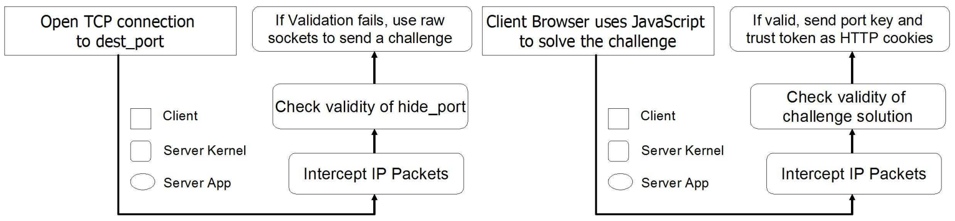

Open TCP connection to dest_port
If Validation fails, use raw sockets to send a challenge
Check validity of hide_port
Intercept IP Packets
Client
Server Kernel
Server App
Client Browser uses JavaScript to solve the challenge
If valid, send port key and trust token as HTTP cookies
Check validity of challenge solution
Intercept IP Packets
Client
Server Kernel
Server App

Fig. 6. Challenge Server Control Flow

Challenge Server. The challenge server operates at the IP layer on the server side. Figure 6 show the detailed control flow of the challenge server. The challenge server facilitates client transparent port key delivery to the client. This is accomplished by ensuring that the URL of the website refers to the challenge server. In other words, a DNS (Domain Name Service) lookup on the URL's domain name would return the IP address of the challenge server. Hence, a client's first access to the website is automatically directed towards the challenge server. Additionally, when a client sends packets with incorrect port numbers, it is redirected to the challenge server. This is required to ensure that clients that experience large clock drifts can continue to access the web server.

The challenge server intercepts the packet at the IP layer. The challenge server uses a C program (operating at the IP layer) to send a cryptographic challenge to the client along with a JavaScript to solve the challenge bundled as a standard webpage. The client-side browser can use the JavaScript to solve the challenge. Note that performing client-side computation can significantly throttle the DoS attackers. We have implemented an adaptive challenge algorithm that is similar to the one described in [Wang and Reiter 2004]. On solving the challenge correctly, the challenge server sends the port key and initial clock skew as standard HTTP cookies to the client-side browser. Further, the challenge server automatically redirects the client to the web server using HTTP redirect. All further requests from the client are forwarded to the web server by the server-side firewall after it verifies the authentication code (destination port number) on the IP packet.

Note that the challenge server cannot be protected using port hiding. Hence, we need to ensure it is very hard to launch DoS attacks on the challenge server. For this purpose, we run the challenge server atop of raw sockets that partially emulate a TCP connection [Kandula et al. 2005] using a stateless TCP/IP server approach [Halfbakery]. This makes our challenge server resilient to TCP level attacks such as SYN floods and SYN+ACK floods that attempt to exhaust the number of open TCP connections on the web server.

Client-side implementation of the challenge solver uses Java applets, while the challenge generator and solution verifier at the server were implemented using C. Although using Java applets is transparent to most client-side browsers (using the standard browser plug-in for Java VM), it may not be transparent to an automated client-side script. However, a client-side script can use its own mechanism to solve the challenge without having to rely on the Java applet framework. Our experiments showed that a client-side challenge solver using a C program and Java applet requires 1 second and 1.1 seconds (respectively) to solve a challenge with hardness m=20. Our client transparent challenge solver is fair to the legitimate clients since the attackers can use any mechanism (including a non-client transparent C program) to solve the challenge.

The challenge server is implemented as a pluggable kernel module that operates at the IP layer (connectionless). Our experiments showed that the challenge server can generate about one million challenges per second and check about one million challenges per second. Given that the challenge server can handle very high request rates and serves only two types of requests (challenge generation and solution verification), it would be very hard for an adversary to launch a DoS attack on the challenge server. Further, one can adaptively vary the cost of solving the challenge by changing the hardness parameter m. For example, setting the challenge hardness parameter $m = 20$ ensures that a client expends one million units (=2^m) of effort to solve the challenge and the server expends only one unit of effort to check a solution's correctness.

5. EVALUATION

In this section, we present two sets of experiments. The first set of experiments studies the effectiveness of admission control using port hiding. The second of experiments demonstrates the effectiveness of trust tokens.

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 [Apache 2005a] 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 [FireFox 2005] running on Linux. The web browser was instrumented to programmatically send requests to the server using JavaScripts [Netscape]. We measured the average client-side throughput in terms of the number of web pages served per second (WPPs). We have also conducted experiments using Microsoft IE 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.

The second service is a database intensive web transaction processing benchmark TPCW 1.0 [TPC 2000]. We used a Java based workload generator from PHARM [PHARM 2000]. We modified the workload generator to handle HTTP tokens. We used Apache Tomcat 5.5 [Apache 2004] as our web server and IBM DB2 8.1 [IBM 2005] 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.

	plain	port hiding	IPSec
TPCW 1 (WIPs)	4.68	4.64 (0.80%)	4.63 (1.0%)
TPCW 2 (WIPs)	12.43	12.41 (0.16%)	12.41 (0.18%)
TPCW 3 (WIPs)	10.04	10.01 (0.30%)	10.00 (0.37%)
HTTPD (WPPs)	100	98 (2.06%)	98 (2.05%)

Table I. Port Hiding: Overhead

nb	HTTP (WPPs)	TPCW 1 (WIPs)	TPCW 2 (WIPs)	TPCW 3 (WIPs)
0	73.7 (26.32%)	4.60 (1.80%)	12.33 (0.80%)	9.92 (1.20%)
1	87.6 (12.35%)	4.62 (1.30%)	12.37 (0.48%)	9.96 (0.76%)
2	94.6 (5.37%)	4.63 (1.00%)	12.40 (0.27%)	9.99 (0.52%)
3	98 (2.06%)	4.64 (0.80%)	12.41 (0.16%)	10.01 (0.30%)

Table II. Port Hiding: Varying *nb*

5.1 Admission Control Filter

We present two sets of experiment on port hiding: (i) measuring the operational overhead of port hiding, and (ii) measuring the resilience of port hiding towards DoS attacks.

5.1.1 *Performance Overhead.* Table I shows the throughput with and without port hiding using $nb = 3$. The numbers in table I are the absolute values, and the numbers in brackets show the percentage drop in throughput due to port hiding. The percentage overhead for TPCW is smaller than for HTTPD. 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 I 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 II 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 II 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 [DARPA 1981]. 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.

5.1.2 *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

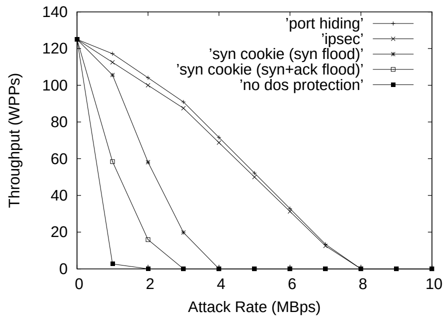


Fig. 7. DoS Attack on HTTPD

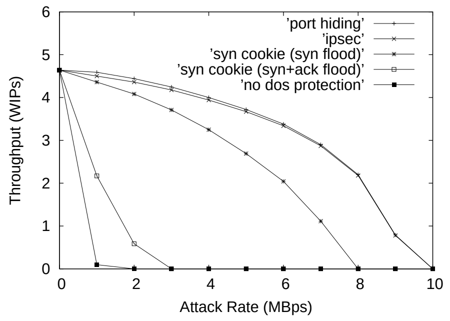


Fig. 8. DoS Attack on TPCW

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 [Savage et al. 2000][Yang et al. 2005] and ingress filtering [Ferguson and Senie 1998].

Figure 7 and Figure 8 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 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.

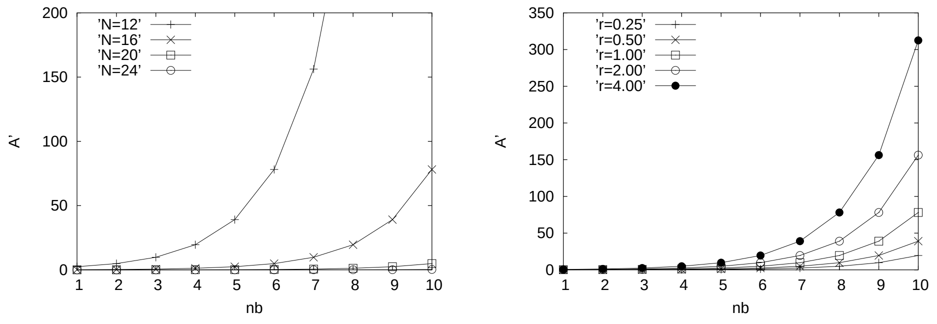
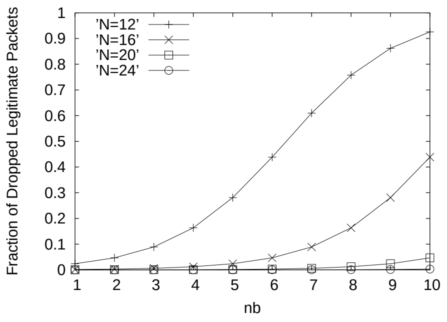


Fig. 9. Guessing the Authentication Code: A' denotes the average number of attackers that have guessed their authentication code at any given time instant

Also the breakdown point for TPCW (9.5 MBps) is 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.



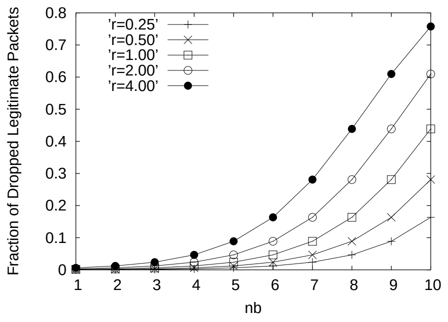


Fig. 10. Fraction of Dropped Legitimate Packets

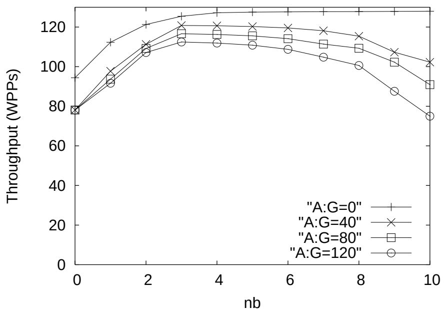


Fig. 11. Application Level Throughput under DoS Attacks for HTTPD

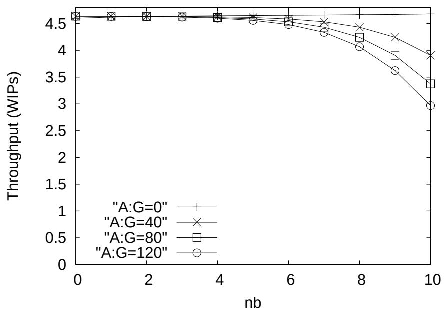


Fig. 12. Application Level Throughput under DoS Attacks for TPCW

5.1.3 *Attacks on Port Hiding Filter.* As described in Section 3.1.2, 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.

Figure 9 shows 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 III. 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.

Figure 10 shows 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 III) is about 82% and 94% respectively of its maximum throughput (throughput is maximum when $A = 0$).

5.2 Port Hiding: Security Analysis

In this section, we present an analytical model for modeling DoS attacks on our port hiding framework. We use our refinement II on port hiding to study DoS attacks (see Section 3.1.2). We summarize the attack scenario as follows. The web server controls the rate parameter r that limits the number of SYN packets accepted from a client per unit time. The adversary on the other hand launches two types of attacks using its available resources. First, the bad clients may use the permissible r SYN packets per unit time to guess their *hide_port*. Second, the adversary may spoof the source IP address of some legitimate client and flood SYN packets at a rate larger than r. From our design in Section 3.1.2, this would coerce the web server to drop packets from a legitimate client. Note that as r increases, more bad clients would be able to guess their correct *hide_port*. On the other hand, as r decreases, the adversary would be able to coerce the web server into dropping packets from more legitimate clients. In the following portions of this section, we

notation	description	default value
N	authentication code size	16 bits
nb	time interval between port number change is 2^{nb} seconds	6
p	fraction of good clients known to the adversary	0.25
G	number of good clients	100
A	number of bad clients	1000
B	aggregate attack bandwidth	100 Mbps
SYN_{size}	size of TCP SYN packet	320 bits

Table III. Notation

variable	description
r	maximum rate of SYN packets permitted by server's firewall
G'	number of good clients attacked by the adversary
A'	number of bad clients that guess their *hide_port*
Z	normalized good client packet drop rate
$ropt$	optimal value of parameter r
$gopt$	optimal value of parameter G'
$aopt$	optimal value of parameter A'
obj	optimal value of the objective Z

Table IV. Variables

use a game theoretic approach to determine the optimal adversarial strategy and the optimal web server strategy for defending against DoS attacks. Our analysis shows the effect of varying important parameters such as: the authentication code size (for small sized authenticators 0-32 bits), varying the time period between authentication code change, the number of bad clients, and the adversary's total bandwidth on the effectiveness of a DoS attack on the server.

In this section we assume that the web server owns separate upstream and downstream bandwidth. Our experiments on the firewall showed that the server can handle about one million packets per second. Note that computing one HMAC-SHA1 [SHA1 2001] using the OpenSSL library [OpenSSL] on a 16-byte input takes less than 1μs. Given that each SYN packet is 320 bits the firewall can handle 320 Mbps before its computing power becomes a bottleneck. One might be able to improve the speed of cryptographic operations (MAC computation) at the firewall by 10-50 times using hardware cryptographic accelerators. Additionally, one can use multiple firewalls to distribute the incoming traffic without requiring any interaction amongst the firewalls. However, care must be taken to ensure that all packets from one client are sent to the same firewall so as to accurately compute the packet rate from that client. This can be easily achieved by routing packets to firewalls based on the prefix of the client's IP address. In our analytical model we assume that the downstream bandwidth to the server becomes a bottleneck before the firewall's computing resources are saturated.

Table III and IV summarize the notation used in our model. The adversary's goal is to partition its resources across two types of attacks: The first part of its bandwidth is used to coerce the web server into dropping packets from a fraction of known legitimate clients. We characterize these two partitions using variables G' and A'. Let $G' \leq pG$ denote the number of legitimate clients whose packets are dropped by the server-side firewall. Let $A' \leq A$ denote the average number of bad clients who have guessed their *hide_port* within one time period. We define the adversary's objective function Z as the rate at which legitimate packets are dropped by the server-side firewall normalized by the capacity of the web server.

Let X denote the number of packets per unit time accepted from each legitimate client when there are G good clients and no bad clients, where $G * X$ denotes the total capacity of the web server. In the presence of an adversary, all the packets from G' clients would be dropped by the server. This amounts to $G' * X$ dropped packets per unit time. Since packets from G' good clients are dropped by the firewall, the effective number of good clients served by the web server is $G - G'$. Further, since A' bad clients have guessed the *hide_port*, they are capable of sending packets through the server-side firewall. In total, we have $G - G' + A'$ clients competing for the server's resources. Note that low level resources (sockets, TCBs) cannot be protected from these A' bad clients. Assuming that the server can fairly distribute its resources among all the admitted $G - G' + A'$ clients, each node gets $\frac{1}{G-G'+A'}$ fraction of the server's resources. Since we have $G - G'$ good clients, the total fraction of server resources available to the good clients is $\frac{G-G'}{G-G'+A'}$. Hence, the total packet rate accepted from good clients is $\frac{G-G'}{G-G'+A'} * G * X$, where $G * X$ is the capacity of the web server. Recall that the server accepted a packet rate of $G * X$ from good clients when there is no DoS attack on the web server. This amounts to $G * X - \frac{G-G'}{G-G'+A'} * G * X = \frac{A'}{G-G'+A'} * G * X$ dropped packets per unit time. Hence, the drop rate for good clients is $G' * X + \frac{A'}{G-G'+A'} * G * X$. We normalize the packet drop rate by a factor $G * X$ to derive the normalized packet drop rate Z in Equation 1.

$$Z = \frac{A'}{G - G' + A'} + \frac{G'}{G} \tag{1}$$

Given the adversarial objection function Z, the game theoretic formulation of this problem is as shown in Equation 2. The adversary controls the parameter G', subject to the following two constraints. Since the adversary knows the IP address of only pG good clients, $G' \leq pG$. The amount of bandwidth required to attack only good client is $SYN_{size} * r$ bits per second (bps). Recall that if the adversary sends r SYN packets per second with the source IP address spoofed using a good client C's IP address, then it is very likely that the firewall would assume that the client C is malicious. Hence, to attack G' good clients, the adversary requires at least $G' * SYN_{size} * r$ bps. Hence, we have the second constraint $G' * SYN_{size} * r \leq B$.

Now the adversary would use the residual bandwidth, namely $B_{res} = B - G' * SYN_{size} * r$, for discovering the *hide_port* corresponding to one of more bad clients. Using this residual bandwidth, the adversary can send $B_{syn} = \frac{B_{res}}{SYN_{size} * A}$ SYN packets per bad client. Since, the firewall restricts the number of SYN packets from any client to r, it does not help if the adversary has infinite bandwidth (B). The attack rate useful for the adversary is $B_{syn}^{act} = min(B_{syn}, r)$. The total number of SYN packets that the adversary can send on behalf of one bad client during one time period is $B_{syn}^{act} * 2^{nb}$. Given that the size of the authentication code is N bits, the probability that one bad client successfully discovers its *hide_port* is $Pr_{discover} = min\left(1, \frac{B_{syn}^{act} * 2^{nb}}{2^N}\right)$. Hence, the total number of bad clients that discover their *hide_port*s is given by $A' = A * Pr_{discover}$.

$$\begin{aligned} Min_{r \geq 0}\ Max_{G' \leq pG}\ Z &= \frac{A'}{G - G' + A'} + \frac{G'}{G}\ \textit{subject to} \\ G' * SYN_{size} * r &\leq B \\ A' &= A * min\left\{1, \frac{2^{nb} * min\left(\frac{B - G' * SYN_{size} * r}{SYN_{size} * A}, r\right)}{2^N}\right\} \end{aligned} \tag{2}$$

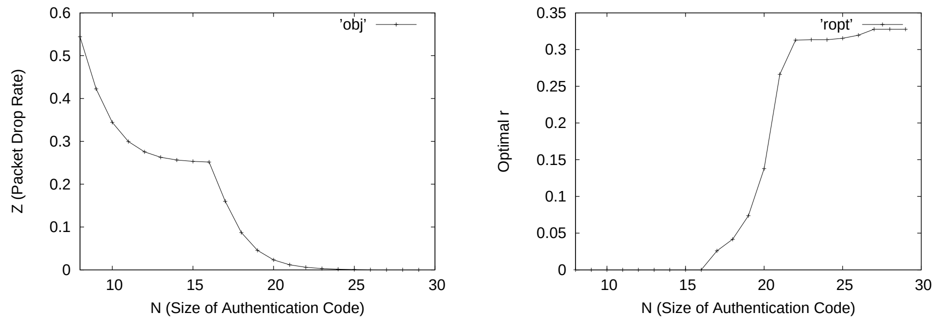


Fig. 13. N: Authentication Code Size

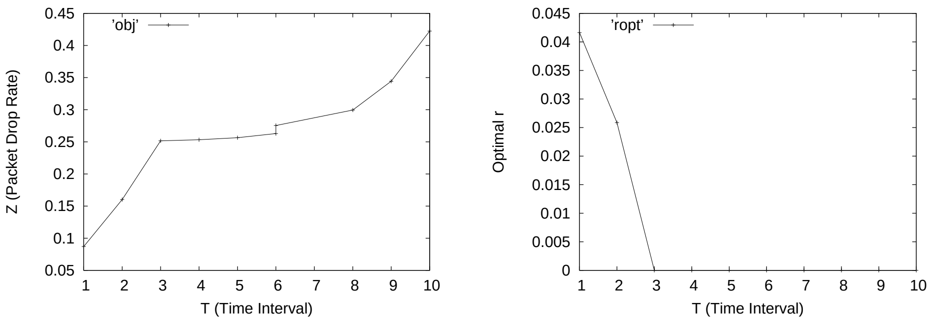


Fig. 14. $T = 2^{nb}$: Time Interval

Solving the game theoretic model in Equation 2 yields the optimal web server filtering parameter *ropt*, the optimal adversarial strategy *gopt* and the optimal packet drop rate *obj* that quantifies the effect of a DoS attack by an adversary on our port hiding framework. Using a game theoretic model allows us to study the worst case adversarial strategy, namely, the attack strategy that causes the maximum damage to the web server. In particular, one should note that if there were multiple adversaries acting independently, then their overall effect would be smaller than that of having one coordinated adversary.

5.2.1 *Analytical Results.* In this section, we present analytical results obtained from our game theoretic model described above. We show the effect of varying several parameters in Table III on the optimal adversarial strategy, the optimal web server strategy and the good client packet drop rate Z that quantifies the damage caused by a DoS attack on the web server. The Y-axis in the Figures 13 to 18 shows the optimal value of the packet drop rate Z (see Equation 1). The Y-axis in the Figures 13 to 18 also show the optimal value for the rate parameter r in million packets per second.

Authentication Code Size N. Figure 13 shows the effect of varying the size of the authentication code N on a DoS attack. Note that *obj* refers to the optimal value of the packet drop rate Z and *ropt* refers to the optimal value of the rate parameter r (see Table IV). When the size of the authentication code N is very

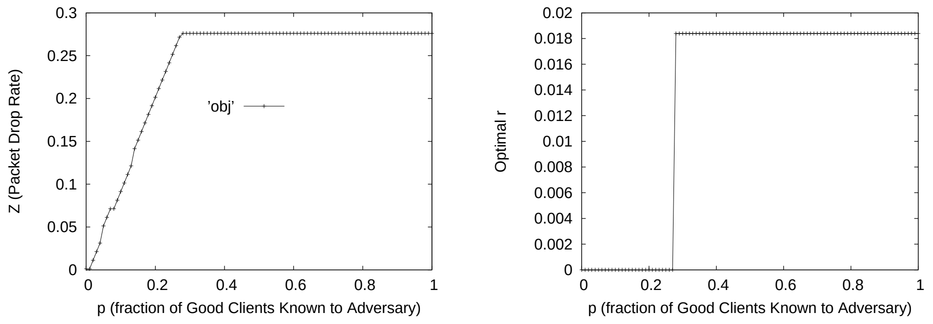


Fig. 15. p: Fraction of Good Clients Known to Adversary

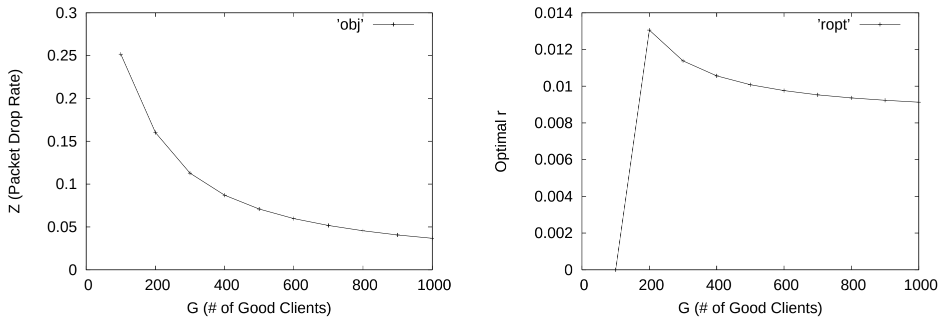


Fig. 16. G: Number of Good Clients

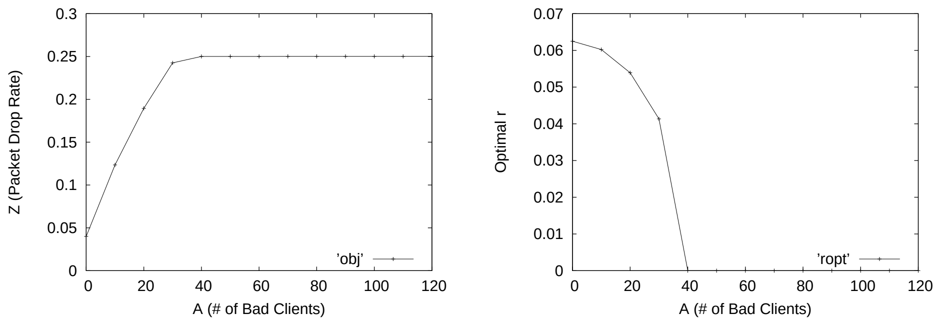


Fig. 17. A: Number of Bad Clients

small, the optimal strategy for the web server is to use a very low value of optimal rate r. Having a small rate r makes it harder for the adversary to guess the authentication code. Nonetheless, when the authentication code size N is small, the adversary would be able to guess the authentication code with non-trivial probability for most bad clients. As the size of the authentication code increases, it becomes harder for the adversary to guess the authentication code. However, having a small rate r would allow the adversary to coerce the web server into denying service to good clients whose IP addresses are known to the adversary. Hence, as

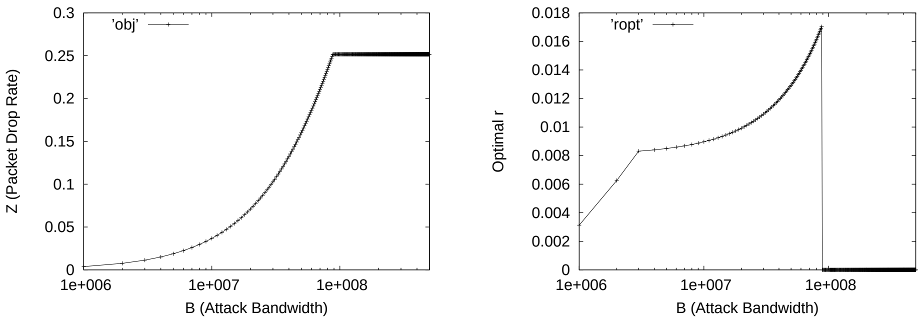


Fig. 18. B: Adversarial Bandwidth

the authentication code size increases, the web server increases r, while not significantly increasing the chance for the adversary to guess the authentication code corresponding to bad clients. Observe from Figure 13 that the optimal value for r is very low when $N \leq 15$; as N increases further, the web server increases r. Observe that the adversary's gain (`obj` in Figure) decreases steeply with N. Observe that with $N \geq 20$ the packet drop rate is almost zero. This demonstrates the ability to use weak authentication codes to defend against DoS attacks.

In addition, these results are further corroborated by our experimental results that show packet drop rate versus N and r in Figures 9 and 10. We measured the precision of our analytical model against the experimental results as follows. We extracted the packet drop rates Z_{expt} (from Figure 10) for parameter settings that match the optimal parameter settings. Using a set of 50 matching parameter settings, we observed that $Z_{analytical}$ was within 11% of Z_{expt} for HTTPD and within 3% for TPCW. We attribute the higher error margin in HTTPD to the fact that our analytical model does not include the overhead due to TCP slow start (which most affects the bandwidth intensive HTTPD application).

Time Interval 2^{nb}. Figure 14 shows the effect of varying nb on a DoS attack. Observe that as we increase nb the legitimate packet drop rate Z increases. A large value of nb allows more time for the adversary to guess the authentication code. However, for bandwidth intensive applications, a small nb results in higher overhead due to TCP slow start. However, our analytical (Figure 14) and experimental results (Figures 9 and 10) show that one can achieve low packet drop rates for large values of nb by suitably decreasing the parameter r. In this figure (and those that follow), one should note that r never actually becomes zero. Observe that if $r = 0$, then no good client (for that matter no client) would be able to send a SYN-packet to the web server. However, a small value of r (say, $r = 1$) is sufficient for an authorized client to establish a connection with the Web server.

Fraction of Good Clients Known to the Adversary p. Figure 15 shows the effect of varying the fraction of good nodes known to an adversary p on a DoS attack. Observe that the legitimate packet drop rate Z first increases with p and then saturates. As the value of p increases, the optimal value of r becomes very high. Using a high value for r would make it hard for an adversary to coerce the web server into denying service for good clients. An interesting observation that follows from Figure 15 is that even if the adversary knows the IP address of all the good clients, it is no more useful than knowing the IP address of $p = 0.28$ fraction of good clients.

Number of Good Clients G. Figure 16 shows the effect of varying the number of good clients on the packet drop rate. Observe that as the number of good clients increases, the packet drop rate decreases. This property follows from the fair queuing nature of our filter that attempts to distribute a server's resources equally to all admitted clients (or weighted by their priorities). Hence, as the number of good clients increases, their fair share of server resources increases and thus the packet drop rate drops.

Number of Bad Clients A. Figure 17 shows the effect of varying the number of bad clients on the packet drop rate. Observe as the number of bad clients increase, the packet drop rate increases and then saturates. As the number of bad clients increases, the optimal web server parameter r is very small. Hence, this would make it very hard for an adversary to guess the authentication code for a large majority of the bad clients. Hence, only a small number of bad clients are admitted into the system, and thus the fair queuing nature of our filter allocates most of the system resources to the good clients.

Adversarial Bandwidth B. Figure 18 shows the effect of varying the total adversarial bandwidth on the packet drop rate. Observe as the adversarial bandwidth increases, so does the packet drop rate. However, the optimal value for Z does not increase beyond a certain value of adversarial bandwidth B. When the adversarial bandwidth becomes very large, the optimal value for the web server parameter r is very small. Observe from Equation 2 the factor $min\left(\frac{B-G'*SYN_{size}*r}{SYN_{size}*A}, r\right)$ evaluates to r for all sufficiently large values of B.

5.2.2 *Summary of Main Results.* We summarize the main results obtained from our experiments and analysis as follows:
(i) Our analysis shows that using 14-18 bit authentication codes may be sufficient to defend against attackers with over 100 Mbps of attack bandwidth. The metric Z (legitimate packet drop rate) drops below 0.02 for authentication codes of size greater than or equal to 20 bits. This shows that one can indeed use weak authentication codes to defend against DoS attacks.
(ii) The metric Z (legitimate packet drop rate) increases only marginally when the time period between changes to authentication codes is increased from 8 seconds ($nb = 3$) to 1024 seconds ($nb = 10$). Hence, for network bandwidth intensive applications one can indeed reduce the overhead due to TCP slow start by using larger nb without significantly compromising the Web server.
(iii) Even if the adversary knows the IP-addresses of all the good clients, the Web server can tolerate DoS attacks by suitably setting the parameter r. In our analytical results, we have shown cases where knowing the IP-addresses of all the good clients does not offer the adversary any more advantage than knowing only $p = 0.28$ fraction of the good client IP-addresses.
(iv) The port hiding framework can handle arbitrarily powerful adversaries (with high attack bandwidth) under the following conditions: (i) The firewall's network

	No Filters	Pre-auth	IPSec
TPCW 1 (in WIPs)	4.68	4.67 (0.11%)	4.63 (1.11%)
TPCW 2 (in WIPs)	12.43	12.42 (0.06%)	4.67 (0.18%)
TPCW 3 (in WIPs)	10.04	10.04 (0.03%)	10.00 (0.37%)
HTTPD (in WPPs)	100	100 (0.5%)	71.75 (3.2%)
	Challenge	IP level tt Filter	App level tt Filter
TPCW 1 (in WIPs)	1.87 (60%)	4.63 (1.11%)	4.59 (1.92%)
TPCW 2 (in WIPs)	9.35 (24.8%)	12.37 (0.49%)	12.32 (0.89%)
TPCW 3 (in WIPs)	6.19 (38.3%)	9.98 (0.61%)	9.91 (1.33%)
HTTPD (in WPPs)	0.3 (99.7%)	97.5 (2.4%)	96.25 (3.7%)

Table V. Overhead

Servlet Name	Admin Req	Admin Resp	Best Seller	Buy Conf	Buy Req	Exec Search	Home
Latency (ms)	2.87	4666.63	2222.09	81.66	5.93	97.86	2.93
Frequency	0.11	0.09	5.00	1.21	2.63	17.20	16.30
Utility	0	0	3	10	4	0	0
Servlet Name	New Prod	Order Disp	Order Inq	Prod Detail	Search Req	Shop Cart	
Latency (ms)	14.41	9.75	0.70	0.88	0.55	0.83	
Frequency	5.10	0.69	0.73	18.00	21.00	11.60	
Utility	0	2	1	1	0	2	

Table VI. TPCW Servlet Mean Execution Time (ms), Servlet Execution Frequency (percentage) and Servlet Utility

$S1$	always attack
$S2$	behave good and attack after reaching the highest Priority level

Table VII. Attack Strategies

$T1$	request flooding
$T2$	low utility requests
$T3$	old tt
$T4$	invalid tt

Table VIII. Attack Types

$A1$	HTTPD
$A2$	TPCW

Table IX. Applications

bandwidth is not completely choked by the adversary. (ii) The firewall's computing power is not a bottleneck. Our experiments on the port hiding filter showed that it can handle over one million packets per second. Given that each SYN packet is 320 bits the firewall can handle 320 Mbps before its computing power becomes a bottleneck. One can further use hardware cryptographic accelerators to improve this throughput by 10-50 times.

5.3 Congestion Control Filter

We have so far studied the effectiveness of port hiding against DoS attacks. In particular, we assumed that the attackers would send the same type of requests as a good client. In this section, we demonstrate the lack of current mechanisms in defending against application level DoS attacks, wherein attackers use cleverly crafted resource intensive requests, and show the effectiveness of our congestion control filter in mitigating them. In rest of this section, we present two sets of experiments on trust tokens. The first set of experiments measures the performance overhead, and the second set of experiments demonstrates the resilience of trust tokens to application level DoS attacks.

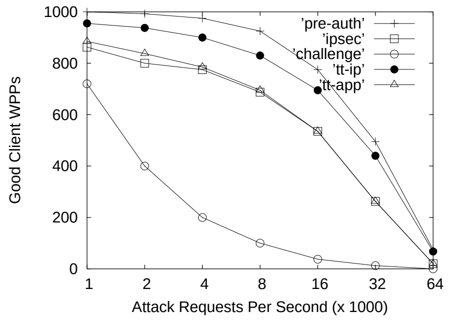


Fig. 19. $\langle S1, T1, A1\rangle$

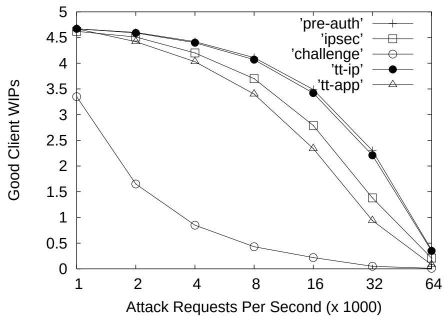


Fig. 20. $\langle S1, T1, A2\rangle$

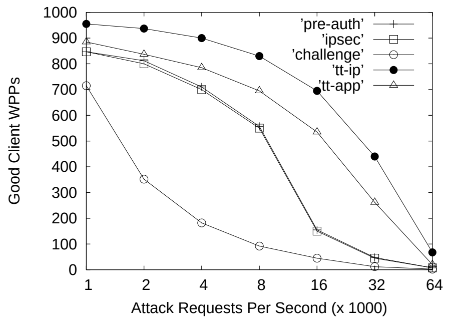


Fig. 21. $\langle S1, T2, A1\rangle$

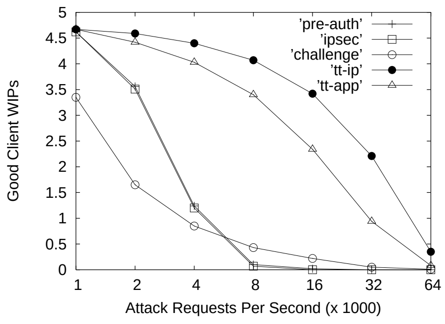


Fig. 22. $\langle S1, T2, A2\rangle$

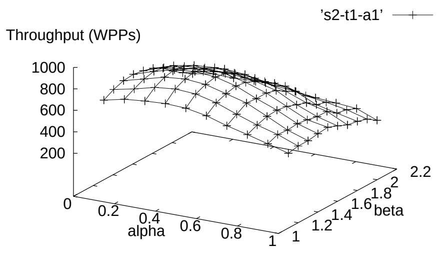


Fig. 23. $\langle S2, T1, A1\rangle$

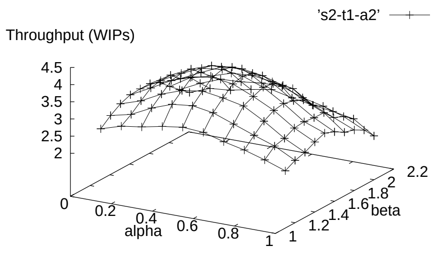


Fig. 24. $\langle S2, T1, A2\rangle$

5.4 Performance Overhead

Table V compares the overhead of our DoS filter ('tt') with other techniques. 'pre-auth' refers to a technique wherein only a certain set of client IP addresses are alone preauthorized to access the service. The 'pre-auth' filter filters packets based on the packet's source IP address. 'IPSec' refers to a more sophisticated preauthorization technique, wherein the preauthorized clients are given a secret key to access the service. All packets from a preauthorized client are tunneled via IPSec using the shared secret key. The 'pre-auth' and 'IPSec' filters assume that all preauthorized clients are benign. Recall that the trust token approach does not require clients to be preauthorized and is thus more general than 'pre-auth' and 'IPSec'. Nonetheless, Table V shows that the overhead of our trust token filter is comparable to the overhead of the less general 'pre-auth' and 'IPSec' approaches. The cryptographic challenge mechanism has significantly higher overhead than the other approaches since it requires both the good and the bad clients to solve cryptographic puzzles each time they send a HTTP request to the server.

We also experimented with two implementations of the trust token filter: 'tt-ip' uses an IP layer implementation of the trust token filter, while 'tt-app' uses an application layer implementation of the same. 'tt-ip' offers performance benefits by filtering requests at the IP layer, while 'tt-app' offers the advantage of not modifying the server side kernel. Table V shows that the overhead of these two implementations are comparable; however, in section 5.5 we show that 'tt-ip' offers better resilience to DoS attacks.

5.5 Resilience to DoS Attacks

In this section, we study the resilience of our trust token filter against application level DoS attacks. We characterize an attack scenario along three dimensions: attack strategy S (Table VII), attack type T (Table VIII) and application A (Table IX). In Table VIII, `request flooding` models traditional DoS attacks wherein the attackers flood the server with requests, `low utility requests` model application level DoS attacks wherein an attacker sends a small number of resource intensive requests, `old tt` models the scenario wherein an attacker initially behaves well to attain a high property and then launches a DoS attack with older trust tokens included in its HTTP requests, and `invalid tt` models the scenario wherein the adversary sends random bits as the trust token. The attack scenarios include all the elements in the cross product $S \times T \times A$. For example, a scenario $\langle S1, T2, A1\rangle$ represents: `always attack` using `low utility requests` on `Apache HTTPD`. Note that these attacks cannot be implemented using standard well-behaved web browsers. Nonetheless, an adversary can use a non-standard malicious browser or browser emulators to launch these attacks.

For experimental purposes, we have assigned utilities to different TPCW servlets based on the application's domain knowledge (see Table VI). For HTTPD we assign utilities to the static web pages as follows. We assume that the popularity of the web pages hosted by the server follows a Zipf like distribution [Yang and Garcia-Molina 2002]. We assign the utility of a request to be in proportion to the popularity of the requested web page. A legitimate client accesses the web pages according to their popularity distribution. However, DoS attackers may attempt to attack the system by requesting unpopular web pages. In a realistic scenario, low popularity web pages are not cached in the server's main memory and thus require an expensive disk I/O operation to serve them. Further, the adversary may succeed in thrashing the server cache by requesting low popularity web pages.

Trust token filter is resilient to the `always attack` strategy: $\langle S1, T1, A1\rangle$ and $\langle S1, T1, A2\rangle$. Figures 19 and 20 show the performance of our trust token filter under the attack scenarios $\langle S1, T1, A1\rangle$ and $\langle S1, T1, A2\rangle$ respectively. For preauthorization based mechanisms this experiment assumes that only the good clients are preauthorized. In a realistic scenario, it may not be feasible to a priori identify the set of good clients, so the preauthorization based mechanism will not always be sufficient. If a bad client always attacks the system (strategy $S1$) then performance of the trust token filter is almost as good as the performance of preauthorization based mechanisms ('pre-auth' and 'IPSec'). This is because, when a client always misbehaves, its priority level would drop to level zero, at which stage all requests from that client are dropped by the server's firewall. Note that with 64K attack requests per second all the DoS filters fail. The average size of our HTTP requests was 184 Bytes; hence, at 64K requests per second it would consume 94.2 Mbps thereby exhausting all the network bandwidth available to the web server. Under such bandwidth exhaustion based DoS attacks, the server needs to use network level DoS protection mechanisms like IP trace back [Savage et al. 2000][Yang et al. 2005] and ingress filtering [Ferguson and Senie 1998].

Trust token filter is resilient to application level DoS attacks: $\langle S1, T2, A1\rangle$ and $\langle S1, T2, A2\rangle$. Table VI shows the mean execution time for all TPCW servlets. Some servlets like 'admin response' and 'best seller' are expensive (because they involve complex database operations), while other servlets like 'home' and 'product detail' are cheap. Figures 21 and 22 show an application level attack on HTTPD and TPCW respectively. In this experiment we assume that only 10% of the preauthorized clients are malicious. Figures 21 and 22 show the inability of network level filters to handle application level DoS attacks and demonstrate the superiority of our trust token filter. One can also observe from figures 21 and 22 that HTTPD can tolerate a much larger attack rate than TPCW. Indeed, the effectiveness of an application level DoS attack on a HTTPD server serving static web pages is likely to be much lower than a complex database intensive application like TPCW.

Several key conclusions that could be drawn from Figures 19, 20, 21 and 22 are as follows: (i) 'IPSec' and 'pre-auth' work well only when preauthorization for all clients is acceptable and if all preauthorized clients are well behaved. Even in this scenario, the performance of 'tt-ip' is comparable to that of 'IPSec' and 'pre-auth'. (ii) Even if preauthorization for all clients is acceptable and a small fraction (10% in this example) of the clients are malicious, then 'IPSec' and 'pre-auth' are clearly inferior to the trust token filter. (iii) If preauthorization for all clients is not a feasible option then 'IPSec' and 'pre-auth' do not even offer a valid solution, while the trust token filter does. (iv) The challenge based mechanisms incur overhead on both good and bad clients and thus significantly throttle the throughput for the good clients as well, unlike the trust token filter that selectively throttles the throughput for the bad clients.

5.6 Attacks on Trust Token Filter

In Section 5.5 we have studied the resilience of the trust token filter against DoS attacks. In this section, we study attacks that target the functioning of the trust token filter.

Additive increase and multiplicative decrease parameters α and β: $\langle S2, T1, A1\rangle$ and $\langle S2, T1, A2\rangle$. Figures 23 and 24 show the throughput for a good

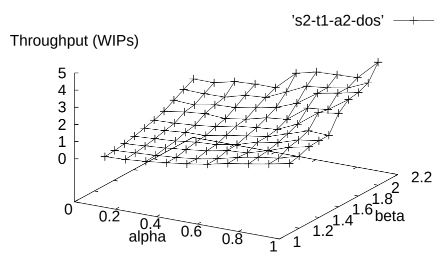


Fig. 25. $\langle S2, T1, A2 \rangle$

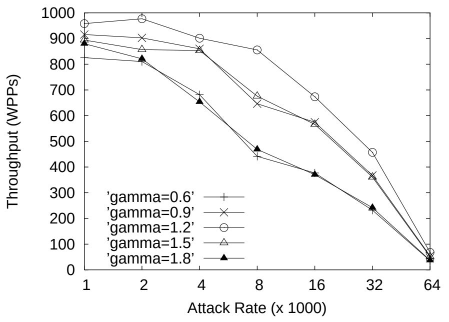


Fig. 26. $\langle S2, T2, A1 \rangle$

client for various values of α and β using applications HTTPD and TPCW respectively. Recall that α and β are the parameters used for the additive increase and multiplicative decrease policy for updating a client's priority level (see Section 3.2). The strategy $S2$ attempts to attack the trust token filter by oscillating between behaving well and attacking the application after the adversary attains the highest priority level. The figures show that one can obtain optimal values for the filter parameters α and β that maximize the average throughput for a good client. Note that the average throughput for a client is measured over the entire duration of the experiment, including the duration in which the adversary behaves well to obtain a high priority level and the duration in which the adversary uses the high priority level to launch a DoS attack on the web server. For HTTPD these optimal filter parameters ensure that the drop in throughput is within 4-12% of the throughput obtained under scenario $\langle S1, T2 \rangle$; while the drop in throughput for TPCW is 8-17%. These percentiles are much smaller than the drop in throughput using preauthorization or challenge based DoS protection mechanisms (see Figures 21 and 22).

Figure 25 shows the average client throughput when the adversary is launching a DoS attack on the web server. When the application is under a DoS attack, large values of α and β maximize the throughput for a good client. Note that a large α boosts the priority level for good clients while a large β penalizes the bad clients heavily. This suggests that one may dynamically vary the values of α and β depending on the server load.

Server resource utilization parameter γ: $\langle S2, T2, A1 \rangle$ and $\langle S2, T2, A2 \rangle$. Figures 26 and 27 show the average throughput for the good clients under the scenario $\langle S2, T2, A1 \rangle$ and $\langle S2, T2, A2 \rangle$ respectively. These experiments show the effect of varying the trust token filter parameter γ. Recall that we use the parameter γ to weigh a request's response time against the request's utility (see Section 3.2). If γ is very small, the filter ignores the response time which captures the amount of server resources consumed by a client's request. On the other hand, if γ is large, the utility of a request is ignored. This would particularly harm high utility requests that are resource intensive. For instance, a high utility request like 'buy confirm' has a response time that is significantly larger than the median servlet response times (see Table VI). The figures show that one can obtain optimal values for the

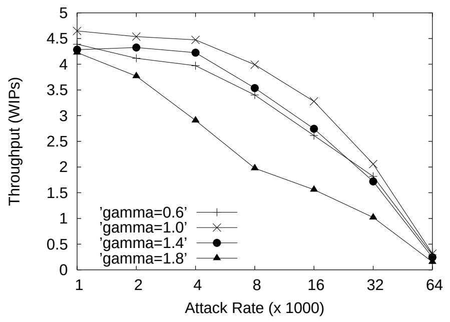


Fig. 27. $\langle S2, T2, A2\rangle$

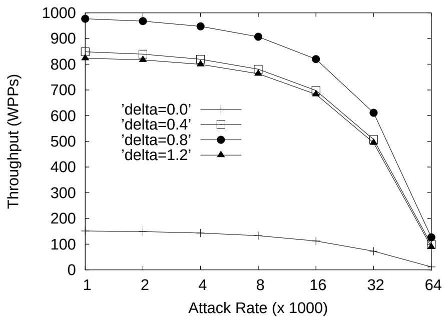


Fig. 28. $\langle S2, T3, A1\rangle$

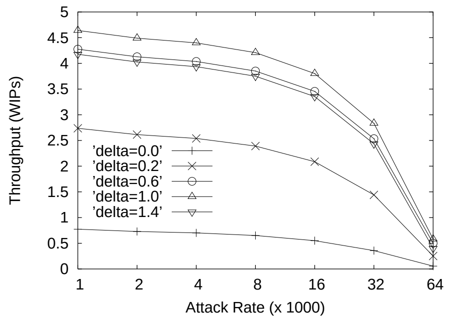


Fig. 29. $\langle S2, T3, A2\rangle$

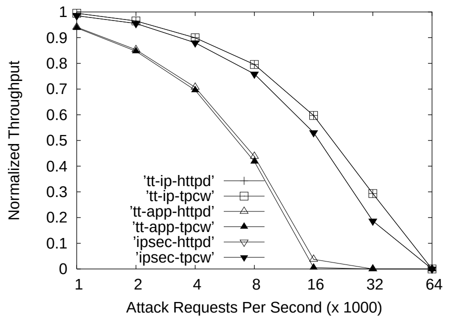


Fig. 30. $\langle S1, T4, A1\rangle$ and $\langle S1, T4, A2\rangle$

filter parameter γ that maximizes the average throughput for a good client. The optimal value for parameter γ ensures that the drop in throughput for HTTPD and TPCW are within 7-11% of throughput measured under scenario $\langle S1,\ T2\rangle$.

Attacking the trust token filter using old trust tokens: $\langle S2, T3, A1\rangle$ and $\langle S2, T3, A2\rangle$. Figures 28 and 29 shows the resilience of our trust token filter against attacks that use old trust tokens. An attacker uses strategy $S2$ to behave well and thus obtain a token with high priority level. Now, the attacker may attack the server using this high priority old token. These experiments capture the effect of varying the trust token filter parameter δ, which is used to penalize (possibly) old trust tokens. A small value of δ permits attackers to use older tokens while a large value of δ may result in rejecting requests even from well behaving clients. The figures show that one can obtain optimal values for the filter parameter δ that maximize the average throughput for a good client. Using the optimal value for parameter δ we observed that the drop in throughput for HTTPD and TPCW is within 3-7% of throughputs measured under scenario $\langle S1,\ T2\rangle$.

Attacking the filter using invalid (spoofed) trust tokens: $\langle S1, T4, A1\rangle$ and $\langle S1, T4, A2\rangle$. Figure 30 shows the effect of attacking the trust token filter by sending invalid cookies for both HTTPD and TPCW. Note that if the verification process for the trust token were to be expensive, then an attacker can launch a DoS attack directly on the verification process itself. We have already shown in Table V that the overhead of our trust token filter is comparable to that of the network layer DoS filters. This experiment shows that the drop in throughput on sending invalid tokens is comparable to sending packets with invalid authentication headers using IPSec. Observe from the figure that the drop in throughput for the IP layer implementation of the trust token filter and IPSec is the same for both the applications HTTPD and TPCW. Observe also that the throughput for the application layer implementation of the trust token filter ('tt-app') is significantly poorer than the IP layer implementation ('tt-ip'). Also, the application layer implementation for HTTPD and TPCW show slightly different impact on the throughput primarily because Apache HTTPD filters (written in 'C') are faster than Apache Tomcat filters (written in 'Java').

Based on our experiments on HTTPD and TPCW we recommend the following values for parameters α, β, γ and δ. For HTTPD: α = 0.3±0.1, β = 1.5±0.2, γ = 1.2±0.1, δ = 0.8±0.2; For TPCW: α = 0.6±0.1, β=1.5±0.3, γ = 1.0±0.1; δ = 1.0±0.2. Automated techniques for determining suitable parameters based on workload characteristics are outside the scope of this paper.

5.7 Integrated DoS Protection

So far we have demonstrated the effectiveness of our admission control filter and the congestion control filter. In this section, we present an evaluation of our system that contains both these filters: the congestion control filter (ccf) layered on top of the admission control filter (acf). The integrated acf + ccf filter implements an additional optimization: when a client consistently has low priority, the ccf filter notifies the acf filter to provide no future port keys to that client.

In this experiment we assume that there are two types of attackers: admitted attackers (attackers who have been granted valid port keys) and unadmitted attackers. The admitted attackers attempt to launch application level DoS attacks on the web server (which bypass the admission control filter and must be handled by the congestion control filter), while the rest launch network level DoS attacks (which may be handled by the admission control filter). While the admission control filter blocks network traffic from unadmitted attackers, it is vulnerable to application level DoS attacks. On the other hand, the congestion control filter (when not layered on top of the admission control filter) is vulnerable to network level DoS attacks from unadmitted attackers. On the other hand, the integrated system proposed in this paper offers superior DoS protection against a wide range of attackers. In this section, we compare our filters with IPSec; other DoS protection mechanisms described earlier in the paper all perform worse than IPSec and hence are not shown in Figures 31-34.

Table X shows the performance overhead of these DoS protection filters. We observe that the overhead of the acf + ccf filter is lower than the combined overhead of the acf and the ccf filters. The integrated acf + ccf filter is implemented such that they share the fair queuing filter; hence, a client's traffic incurs the fair queuing overhead only once. Nonetheless, the aggregate overhead of our filters is less than 3.17%. While IPSec offers lower performance overhead, we demonstrate the superiority of the acf + ccf filter in defending against DoS attacks.

Figures 31 and 32 show application level throughput when the web server is

filter	HTTPD (WPPs)	TPCW 1 (WIPs)	TPCW 2 (WIPs)	TPCW 3 (WIPs)
IPSec	98 (2.05%)	4.63 (1.0%)	12.41 (0.18%)	10.00 (0.37%)
acf	98 (2.06%)	4.64 (0.80%)	12.41 (0.16%)	10.01 (0.30%)
ccf	97.6 (2.4%)	4.62 (1.11%)	12.37 (0.48%)	9.98 (0.61%)
acf + ccf	96.8 (3.17%)	4.60 (1.70%)	12.35 (0.60%)	9.96 (0.82%)

Table X. Performance Overhead

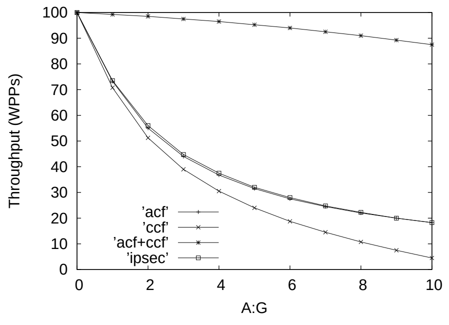


Fig. 31. HTTPD with 50% admitted attackers: IPSec and acf nearly overlap

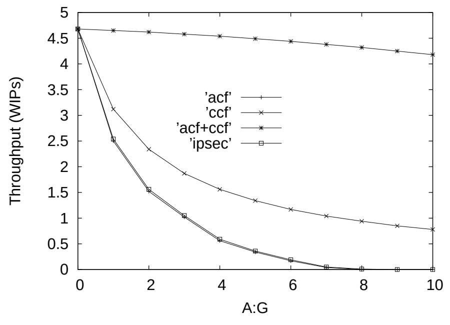


Fig. 32. TPCW with 50% admitted attackers: IPSec and acf nearly overlap

under DoS attacks for different values of $A : G$ (ratio of the number of attackers to the number of legitimate clients) assuming that 50% of the attackers are admitted. Evidently, the acf filter is vulnerable to application level DoS attacks and the ccf filter is vulnerable to network level DoS attacks; while the acf + ccf filter can tolerate a wide range of DoS attacks effectively. However, we observe for the HTTPD application that the acf filter performs better than the ccf filter; the converse holds for the TPCW application. HTTPD being a bandwidth intensive application is more vulnerable to network level DoS attacks and is thus more affected in the absence of the acf filter. TPCW being a compute (database) intensive application is more vulnerable to application level DoS attacks and is thus more affected in the absence of the ccf filter. The figures also indicate that IPSec is only as good as the acf filter despite violating client transparency.

Figures 33 and 34 show the application level throughput when the web server is under DoS attacks with $A : G = 5$. However, we vary the percentage of attackers who are admitted; the admitted attackers launch application level DoS attacks, while the rest launch network level DoS attacks. Observe that as the percentage of admitted attackers increases, the effectiveness of the acf filter drops and that of the ccf filter increases. Using the acf + ccf filter, the throughput drop for HTTPD is 4% and that for TPCW is 13% when 100% of the attackers are admitted. This is primarily because application level DoS attacks on a complex 3-tier application like TPCW are significantly more disruptive than those on a simple HTTPD application.

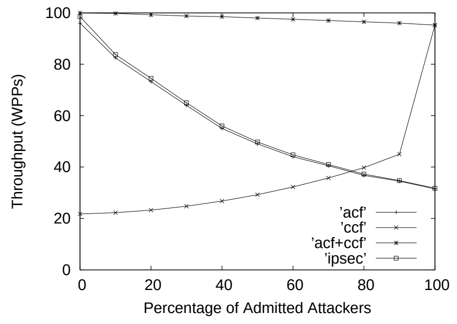


Fig. 33. HTTPD: Varying the % of Admitted Attackers

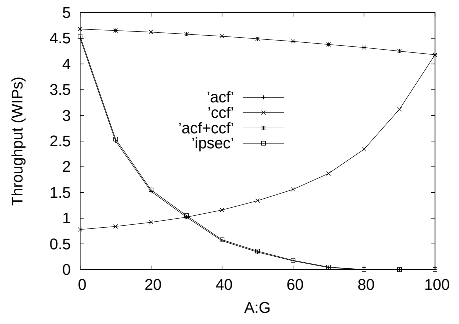


Fig. 34. TPCW: Varying the % of Admitted Attackers

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 [Egevang and Francis 1994] 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 [Wang and Reiter 2004] 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.

Non-HTTP Applications. The methodologies proposed in this paper apply to a wide range of applications. However, our client transparent implementation operates only on HTTP. Non-HTTP applications (such as DNS, ARP, etc) do not support client-side programmability (say, scripting); in such cases, one has to sacrifice client transparency.

Dynamic and Mobile IP. In the case of dynamic IP a client's IP address will remain the same throughout a session. Assuming that the client does not often connect and reconnect to the network, the overhead due to challenge mechanism can be quite small. Using mobile IP allows a client to use its permanent IP address (home address) even when mobile user joins another network (care of address). Hence, mobile IP users can seamlessly inter-operate with our proposed solution.

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 [Dierks and Allen] or IPSec [Kent 1998][Yin and Wang 2005] 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 in higher layers in the networking stack that permits an attacker to attack lower layers in the networking stack. Furthermore, 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 light weight but requires changes to the client-side kernel and requires superuser privileges at the client. Our approach simultaneously satisfies client transparency, and yet presents a light weight IP level authentication mechanism.

There are several network level DoS protection mechanisms including IP trace back [Savage et al. 2000], ingress filtering [Ferguson and Senie 1998], SYN cookies [Bernstein 2005] and stateless TCP server [Halfbakery] to counter bandwidth exhaustion attacks and low level OS resource (number of open TCP connections) utilization attacks. Yang et al.[Yang et al. 2005] proposes a cryptographic capability based packet marking mechanism to filter out network flows from DoS attackers. There are several low level DoS protection mechanisms includes SYN tokens [Bernstein 2005] and stateless TCP server [Halfbakery] to handle bandwidth exhaustion and low level OS resource (number of open TCP connections) utilization attacks. 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.

The paper [Savage et al. 2000] suggests using IP traceback to defend against spoofed source IP-address based DoS attacks. Their solution requires the IP routers to probabilistically mark IP packets. One could use ingress filtering [Ferguson and Senie 1998] to filter attack packets close to the origin of the attack (at the attackers ISP). However, IP traceback is not effective against DDoS attacks wherein an attacker compromises a large number of hosts scattered all over the Internet.

However none of the above network level DoS protection techniques are capable of addressing application layer DoS attacks. Our experiments show that the inability of a challenge based mechanism to selectively throttle the performance of the bad clients can significantly harm the performance for the good clients. Crosby and Wallach [Crosby and Wallach 2003] present DoS attacks that target application level hash tables by introducing collisions. We have described other examples of application level DoS attacks in Section 2.

Jung et al. [Jung et al. 2002] characterizes the differences between flash crowds and DoS attacks. The paper proposes to use client IP address based clustering and file reference characteristics to distinguish legitimate requests from the DoS attack requests.

Siris et al. [Siris and Papagalou 2004] suggests using request traffic anomaly detection to defend against DoS attacks. We have shown in Section 2 that an application level DoS attack may mimic flash crowds, thereby making it hard for the server to detect a DoS attacker exclusively using the request traffic characteristics.

Recently, several web applications (including Google Maps [Google b] and Google Mail [Google a]) have adopted the Asynchronous JavaScript and XML (AJAX) model [Wei 2005]. 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.

Several papers have presented techniques for implementing different QoS guarantees for serving web data [Chandra et al. 2000][Cherkasova and Phaal 2002][Cardellini et al. 2002]. A summary of past work in this area is provided in [Iyengar et al. 2005]. These papers are not targeted at preventing DoS attacks and do not discuss application level DoS attacks.

The paper [Xu and Lee 2003] suggests a game-theoretic approach to model the relationship between a DoS adversary and the web service as a two player game.

7. CONCLUSION

In this paper we have proposed application-specific client-transparent mechanisms to handle DoS attacks. First, we perform admission control using port hiding to limit the number of concurrent clients being served by the online service. Second, we perform congestion control on admitted clients to allocate more resources to good clients by adaptively setting a client's priority level in response to the client's requests, in a way that incorporates application-level semantics. We have presented a detailed evaluation using two sample applications: a bandwidth intensive Apache HTTPD and a database intensive TPCW benchmark (running on Apache Tomcat and IBM DB2). Our experiments show that our techniques incur low performance overhead and are resilient to DoS attacks. We have studied several attacks that demonstrate the importance of choosing the port hiding filter parameters and congestion control filter parameters. Our techniques can be easily deployed into existing web/application servers to improve their resilience to DoS attacks.

Our future work is focused along two dimensions. First, we are working on techniques to embed longer authentication codes (more than 16-bits) in a client transparent manner. In particular, we are exploring techniques to embed an authentication code in different parts of the HTTP request message in a way that is resilient to IP level fragmentation. As we have noted in our analysis and experiments, using longer authentication codes is useful in defending against more powerful adversaries. Second, we are exploring techniques to refine our API that allow application programmers to estimate a request's utility and develop sample implementations for different classes of applications (network intensive versus CPU intensive versus database intensive). We are also studying other request profiling techniques in addition to the response time metrics used in this paper.

Acknowledgement: This research was partially supported by NSF CSR and CyberTrust

REFERENCES

Apache. 2004. Apache tomcat servlet/JSP container. http://jakarta.apache.org/tomcat.

Apache. 2005a. Apache HTTP server. http://httpd.apache.org.

Apache. 2005b. Introduction to server side includes. http://httpd.apache.org/docs/howto/ssi.html.

Bernstein, D. J. 2005. SYN cookies. http://cr.yp.to/syncookies.html.

Black, D. RFC 2983: Differentiated services and tunnels. http://www.faqs.org/rfcs/rfc2983.html.

Cardellini, V., Casalicchio, E., Colajanni, M., and Mambelli, M. 2002. Enhancing a web server cluster with quality of service mechanisms. In *Proceedings of 21st IEEE IPCCC*.

CERT. 2004. Incident note IN-2004-01 W32/Novarg.A virus.

Chandra, S., Ellis, C. S., and Vahdat, A. 2000. Application-level differentiated multimedia web services using quality aware transcoding. In *Proceedings of IEEE special issue on QoS in the Internet*.

Cherkasova, L. and Phaal, P. 2002. Session based admission control: a mechanism for web QoS. In *Proceedings of IEEE Transactions on Computers*.

Crosby, S. A. and Wallach, D. S. 2003. Denial of service via algorithmic complexity attacks. In *Proceedings of 12th USENIX Security Symposium, pp: 29-44*.

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

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

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

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

FIPS. Data encryption standard (DES). http://www.itl.nist.gov/fipspubs/fip46-2.htm.

FireFox. 2005. Mozilla firefox web browser. http://www.mozilla.org/products/firefox.

Google. Google mail. http://mail.google.com/.

Google. Google maps. http://maps.google.com/.

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

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

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

Iyengar, A., Ramaswamy, L., and Schroeder, B. 2005. Techniques for efficiently serving and caching dynamic web content. In *Book Chapter in Web Content Delivery, X. Tang, J. Xu, S. Chanson ed., Springer*.

Juels, A. and Brainard, J. 1999. Client puzzle: A cryptographic defense against connection depletion attacks. In *Proceedings of Networks and Distributed Systems Security Symposium (NDSS)*.

Jung, J., Krishnamurthy, B., and Rabinovich, M. 2002. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In *Proceedings of 11th World Wide Web Conference (WWW2002)*.

Kandula, S., Katabi, D., Jacob, M., and Berger, A. 2005. Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds. In *Proceedings of 2nd USENIX NSDI*.

Kent, S. 1998. RFC 2401: Secure architecture for the internet protocol. http://www.ietf.org/rfc/rfc2401.txt.

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

NetFilter. Netfilter/IPTables project homepage. http://www.netfilter.org/.

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

Nichols, K., Blake, S., Baker, F., and Black, D. RFC 2474: Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers. http://www.faqs.org/rfcs/rfc2474.html.

NIST. AES: Advanced encryption standard. http://csrc.nist.gov/CryptoToolkit/aes/.

OpenSSL. Openssl. http://www.openssl.org/.

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

Poulsen, K. 2004. FBI busts alleged ddos mafia. www.securityfocus.com/news/9411.

Savage, S., Wetherall, D., Karlin, A., and Anderson, T. 2000. Practical network support for IP traceback. In *Proceedings of ACM SIGCOMM*.

SHA1. 2001. US secure hash algorithm I. http://www.ietf.org/rfc/rfc3174.txt.

Siris, V. A. and Papagalou, F. 2004. Application of anomaly detection algorithms for detecting SYN flooding attacks. In *Proceedings of IEEE Globecom*.

Srivatsa, M., Iyengar, A., Yin, J., and Liu, L. 2006a. A client-transparent approach to defend against denial of service attacks. In *25th IEEE Symposium on Reliable Distributed Systems (SRDS)*.

Srivatsa, M., Iyengar, A., Yin, J., and Liu, L. 2006b. A middleware system for protecting against application level denial of service attacks. In *7th ACM/IFIP/USENIX Middleware Conference*.

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

Stubblefield, A. and Dean, D. 2001. Using client puzzles to protect TLS. In *USENIX Security Symposium*.

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

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

Waters, B., Juels, A., Halderman, A., and Felten, E. W. 2004. New client puzzle outsourcing techniques for DoS resistance. In *Proceedings of 11th ACM CCS*.

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

Xu, J. and Lee, W. 2003. Sustaining availability of web services under distributed denial of service attacks. In *Proceedings of IEEE Transactions on Computers pp: 195-208, Vol: 52, Issue: 2*.

Yang, B. and Garcia-Molina, H. 2002. Improving search in peer-to-peer networks. In *22nd Conference ICDCS'03*.

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

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