# A Middleware System for Protecting Against Application Level Denial of Service Attacks

**Mudhakar Srivatsa[†], Arun Iyengar[‡], Jian Yin[‡] and Ling Liu[†]**

College of Computing, Georgia Institute of Technology, Atlanta, GA 30332, USA[†]

IBM T. J. Watson Research Center, Yorktown Heights, NY 10598, USA[‡]

{mudhakar, lingliu}@cc.gatech.edu, {aruni, jianyin}@us.ibm.com

**Abstract.** 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 usually target bottleneck resources such as disk bandwidth, database bandwidth, and CPU resources. In this paper we propose server-side middleware to counter application level DoS attacks. The key idea behind our technique is to adaptively vary a client's priority level, and the relative amount of resources devoted to this client, in response to the client's past requests in a way that incorporates application level semantics. Application specific knowledge is used to evaluate the cost and the utility of each request and the likelihood that a sequence of requests are sent by a malicious client. Based on the evaluations, a client's priority level is increased or decreased accordingly. A client's priority level is used by the server side firewall to throttle the client's request rate, thereby ensuring that more server side resources are allocated to the legitimate clients. We present a detailed implementation of our approach on the Linux kernel and evaluate it using two sample applications: Apache HTTPD micro-benchmarks and TPCW. Our experiments show that our approach incurs low performance overhead and is resilient to application level DoS attacks.

## 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 [32] 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 $6^{th}$ 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. 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 [24]. 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 [6][28][32].

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.

**Problem Outline.** 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.

**Our Approach.** In this paper we propose middleware for protecting a website against application level DoS attacks. Our middleware solution carefully divides its operations between the server's firewall and application layer. The firewall component of our middleware is completely application transparent. The application layer component of our middleware exports an application programming interface (API) to the application programmers to improve the website's resilience to application level DoS attacks.

Our DoS protection middleware is functionally different from most traditional DoS filters. Our mechanism does not attempt to distinguish a DoS attack request from the legitimate ones. Instead, our mechanism examines the amount of resources expended by the server in handling a request, rather than the request itself. 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.

**Benefits.** Our approach to guard an online service against application level DoS attacks has several benefits.
**1.** An obvious benefit that follows from the description of our DoS protection mechanism 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.
**2.** 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.
**3.** Our mechanism does not distinguish requests based on the request rate, the packet headers, or the contents of the request. As pointed out earlier (and discussed in Section 2) it is very hard to distinguish an attack request from the legitimate ones using either the request rate or the contents of the request.

**Contributions.** The key contributions of this paper include:
**1.** We propose a request throttling mechanism that allocates more server resources to the legitimate clients, while severely throttling the amount of server resources allocated to the DoS attackers. This 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 provide a simple application programming interface (API) that permits an application programmer to use our DoS protection mechanism.
**2.** The proposed solution does not require the clients to be preauthorized. The absence of preauthorization implies that the server does not have to establish any out of band trust relationships with the client.
**3.** Our proposed solution is client transparent, that is, a user or an automated client side script can browse a DoS protected website in the same way as it browsed an unprotected website. Our DoS protection mechanisms do not require any changes to the client side software or require super user privileges at the client. The clients can seamlessly browse a DoS protected website using any standard web browser that supports HTTP cookies. All instrumentations required for implementing our proposal can be incorporated on the server side.
**4.** We present a detailed implementation of our proposed solution on the Linux kernel and a concrete evaluation using two sample applications: Apache HTTPD benchmark [2] and the TPCW benchmark [37] (running on Apache Tomcat [3] and IBM DB2 [20]). Our experiments show that the proposed solution incurs low performance overhead (about 1-2%) and is resilient to application level DoS attacks.

## 2   Application Level DoS Attacks

In this section, we present two examples of application level DoS attacks. Then, we discuss existing approaches for DoS protection, highlighting the deficiencies of those approaches in defending against application level DoS attacks.

### 2.1   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 [37]. 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 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   Existing Approaches

**Preauthorization.** One way to defend from DoS attacks is to permit only preauthorized clients to access the web server. Preauthorization can be implemented using SSL [30] or IPSec [25] 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, requiring preauthorization may deter clients from using the online service. Also, for an open e-Commerce web site like eBay or Amazon, it may not be feasible to make an exhaustive list of all clients that should be authorized to access the service. Further, it would be very hard to ensure all authorized clients will behave benignly. A DoS attack from a small subset of preauthorized clients

may render the server unusable.

**Challenge Mechanism.** Challenge based mechanisms provide an alternative solution for DoS protection without requiring preauthorization. A challenge is an elegant way to throttle the intensity of a DoS attack. For example, an image based challenge [24] may be used to determine whether the client is a real human being or an automated script. A cryptographic challenge [40] 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.

**Network Level DoS Attacks.** There are several network level DoS protection mechanisms including IP trace back [33], ingress filtering [13], SYN cookies [4] and stateless TCP server [18] to counter bandwidth exhaustion attacks and low level OS resource (number of open TCP connections) utilization attacks. Yang et al.[45] proposes a cryptographic capability based packet marking mechanism to filter out network flows from DoS attackers. However none of these techniques are capable of addressing application level DoS attacks. Nonetheless one should keep in mind that the application level DoS filters only augment the network level DoS filters but do not replace them.

**Application Level DoS Attacks.** The network layer DoS filters cannot handle application level DoS attacks primarily because they lack application level semantics. There have been some proposals that degrade the image/video quality [15][8][21] when the server experiences heavy overload. It is to be noted that such techniques are more effective in protecting the servers from overload than from DoS attacks.

## 2.3 Threat Model

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 the 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 a client whose IP address is 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 amounts of networking and computing resources at its disposal and thus cannot inject arbitrarily large numbers of packets into the IP network. We assume that the adversary can coordinate activities perfectly to take maximum advantage of its resources.

## 3 Trust Tokens

### 3.1 Overview

Figure 1 shows a high level architecture of our proposed solution. Our approach allocates more server resources to good clients, while severely limiting the amount of
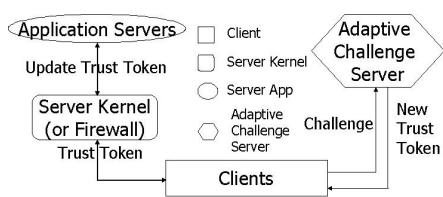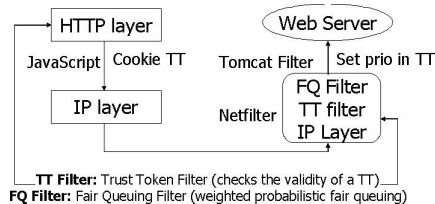
**Fig. 1: Overview**

Application Servers
Update Trust Token
Server Kernel (or Firewall)
Trust Token

Client
Server Kernel
Server App
Adaptive Challenge Server

Adaptive Challenge Server
Challenge
New Trust Token

Clients

Fig. 1: Overview

**Fig. 2: Architecture**

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. 2: Architecture

**Fig. 3: Control Flow**

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

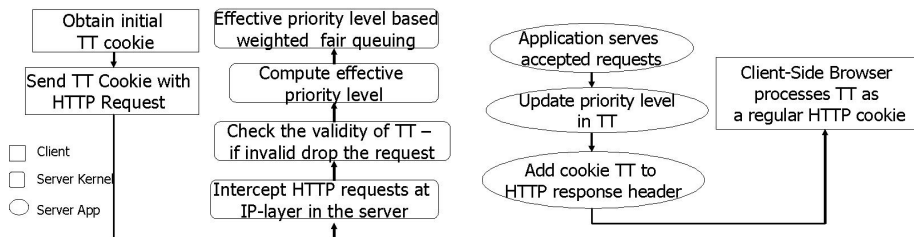Client-Side Browser processes TT as a regular HTTP cookie

Fig. 3: Control Flow

resources expended on DoS attackers. The maximum amount of resources allocated to a client is represented by the client's QoS level. We use *trust tokens* (denoted as TT in Figures 2 and 3) to encode the QoS level that a client is eligible to receive. Although our architecture is capable of supporting arbitrary QoS policies, for the sake of simplicity, we characterize a client's QoS level exclusively by its priority level (a totally ordered numeric value). A client's trust token is embedded in a standard HTTP cookie that is included in all responses from the server to the client. Using the standard 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 dropped at the server's IP layer or firewall.

A client's priority level is used to rate limit its requests at the server's IP layer or the firewall. We use an IP level packet filter to filter HTTP requests from the clients. The packet filter uses weighted fair queuing [35] to throttle a client's request rate based on the client's priority level. Hence, requests from attackers attempting to issue a disproportionately large number of requests (relative to their priority level) are dropped at the IP layer itself. Filtering requests at the IP layer significantly reduces the amount of processor, memory, network, and disk resources expended on that request.

A client's priority level is adaptively varied by the application using application specific semantics and domain knowledge. For this purpose, we provide a simple and flexible API for application programmers. We describe the concrete API with three sample implementations in Section 3.3. 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) request filtering ensures that illegitimate requests are dropped before they can consume much of the server's resources. In this paper we explore several algorithms which could be used to vary the client's priority level and study its effect on the performance of the web server.

Trust tokens are bootstrapped using an initial trust token issued by the challenge server when the client first attempts to access the web server. The trust token is encrypted in such a way that it would be computationally infeasible for a client to unde-

tectably modify the token. We ensure the priority level of a client is kept up to date; since the priority level of a client is continuously varied it is very important for the server to retain the most recent value of a client's priority level, especially, if the client's priority level is dropping.

Our proposed solution is client transparent and requires no changes to the client side software. All our instrumentation is done at the server side thereby making the deployment very easy. The instrumentation at the server side includes:

**Challenge Server.** The challenge server poses a cryptographic challenge to a client, when the client first accesses the website. On correctly solving the challenge, the challenge server is responsible for initializing the client's trust token. A valid trust token allows a client to send requests to the web server. It is important to note that the client does not have to solve a cryptographic challenge every time it sends a request to the server. Further, we use an adaptive mechanism wherein the hardness of solving the challenge depends on the web server's load. Indeed, when the server is not overloaded, our system ensures that the client does not expend its computational resources on solving a challenge before it is granted permission to access the web server.

**Server Kernel or Firewall.** The IP layer at the server is modified to use the client's priority level to filter HTTP requests sent by a client. The priority level is enforced by fair queuing [35] requests at the IP level. Filtering requests at the IP layer saves a lot of computing and memory resources that are otherwise expended on the request as it traverses up the server's network stack.

**Application Server.** The application layer at the server is modified to use application specific rules to update a client's priority level. The client's new priority level is computed using a utility based model that considers the set of recent requests sent by the client and the amount of server resources consumed by these requests.

### 3.2 Design

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

**Trust Token.** A 24 Byte long trust token ($tt$) is constructed as follows: $tt = \langle prio, tv, H_{MK}(cip, sip, tv, prio)\rangle$, 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), $prio$ (4 Bytes) denotes the priority level assigned to the client by the server, $MK$ denotes a secret cryptographic key used by the server and $H$ denotes a keyed pseudo-random function (like HMAC-MD5 or HMAC-SHA1 [27]). A priority level of zero indicates that all requests from the client would be dropped by the server.

**Client Side.** Figure 2 below shows our architecture and Figure 3 shows the operational usage of the trust token. A legitimate client operates as follows. A client obtains its token $tt$ when it solves a challenge. The token is stored as a HTTP cookie in the client's browser. The client includes the token $tt$ in all its HTTP requests to the server.

**Server Side Firewall.** On the server side firewall, we perform two operations. First, we

filter HTTP requests based on the validity of the trust token. Second, if the trust token is valid, the server extracts the client's priority level and throttles the client's request rate using fair queuing.

The server checks if the packet is a HTTP request and if so, it extracts the HTTP cookie $tt$. It validates the trust token $tt$ as follows. A trust token $tt$ is valid if the $tt.cip$ matches the client's IP address, $tt.sip$ matches the server's IP address, and $tt.tv$ is some time in the past ($tt.tv < cur\_time$). If so, the server extracts the priority level ($tt.prio$) from $tt$; otherwise the request is dropped by the firewall.

An adversary may behave benignly until it attains a high priority level and then begin to misbehave. Consequently, the server would issue a trust token with a lower priority level. However, the adversary may send an old trust token (with high priority level) to the server in its future requests. If all responses from the server to the client were tunneled via the firewall, then the firewall can record the client's updated priority level. However, for performance reasons, most application servers are configured in a way that requests are tunneled via the firewall but not the responses [19]. Under such a scenario, we prevent a DoS attack by computing the effective priority level as follows.

The server uses the request's priority level $prio$, the time of cookie issue ($tt.tv$) and the client's request rate $r$ to compute the effective priority level $eprio$ as follows: $eprio = prio * e^{-\delta * max(cur\_time - tt.tv - \frac{1}{r}, 0)}$, where $cur\_time$ denotes the current time. The key intuition here is that if $cur\_time - tt.tv$ is significantly larger than the client's mean inter request arrival time ($\frac{1}{r}$) then the client is probably sending an old trust token. The larger the difference between ($cur\_time - tt.tv$) and $\frac{1}{r}$, the more likely it is that the client is attempting to send an old token. Hence, we drop the effective priority level $eprio$ exponentially with the difference between ($cur\_time - tt.tv$) and $\frac{1}{r}$. Note that the fair queuing filter estimates the client's request rate $r$ for performing weighted probabilistic fair queuing.

Having validated the trust token and extracted its priority level, the server uses $eprio$ to perform weighted probabilistic fair queuing on all incoming HTTP requests from the client. Fair queuing limits the maximum request rate from a client to its *fair share*. Hence, requests from an attacker attempting to send requests at a rate larger than its fair share is dropped by the firewall. We set a client's fair share to be in proportion to its effective priority level ($eprio$). Hence, if a client has a low priority level, then only a very small number of requests from the client actually reach the web server. In the following portions of this section, we propose techniques to ensure that DoS attackers are assigned low priority levels, while the legitimate clients are assigned higher priority levels.

**Server Side Application Layer.** Once the request is accepted by the IP layer packet filter, the request is forwarded to the application. 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 this purpose we use a benefit function $B(rq)$ that estimates the benefit of a client's request $rq$. The benefit of a request takes into account the utility of the request and the resources expended in handling that request. For instance, if the request $rq$ is a credit card transaction, then the utility of request $rq$ could be the monetary profit the server obtains from the transaction. We also define a priority update function $G$ that updates the priority level of a client based on the benefit $B(rq)$.

In our first prototype, we propose to use a utility based benefit function $B(rq) = F(rt, ut)$, where $rq$ denotes the client's request, $rt$ is the time taken by the server to generate a response for the request $rq$, and $ut$ denotes the utility of $rq$. We use a simple benefit function $B(rq) = ut - \gamma * rt$, where $\gamma$ is a tunable parameter. The response time $rt$ is used as a crude approximation of the effort expended by the server to handle the request $rq$. Observe that in computing the benefit $B(rq)$, the response time $rt$ (that denotes the effort expended by the server) is subtracted from the request's utility $ut$.

The new priority level $nprio$ could be computed as $nprio = G(eprio, B(rq))$, where $eprio$ is the current effective priority level of the client. In our first prototype, we use an additive increase and multiplicative decrease strategy to update the priority level as follows: If $B(rq) \geq 0$, then $nprio = eprio + \alpha * B(rq)$, and $nprio = \frac{eprio}{\beta * (1 - B(rq))}$ otherwise. The additive increase strategy ensures that the priority level slowly increases as the client behaves benignly; while the multiplicative decrease strategy ensures that the priority level drops very quickly upon detecting a DoS attack from the client.

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

### 3.3 Implementation

**Client Side.** Our implementation neither requires changes to the client side software nor requires super user privileges at the client. We implement trust tokens using standard HTTP cookies. Hence, the client can use standard web browsers like Microsoft IE or FireFox to browse a DoS protected website in the same manner that it browses an unprotected website. An automated client side script with support for handling HTTP cookies is assumed; such scripts are commonly used on the web today.

**Server Side IP Layer.** On the server side, we use NetFilters [1] for filtering requests at the IP layer. NetFilters is a framework inside the Linux kernel that enables packet filtering, network address translation and other packet mangling. We use NetFilters to hook onto packet processing at the IP layer. Given an IP packet we check if it is a HTTP request and check if it has the $tt$ cookie in the HTTP request header. If so we extract the trust token $tt$, check its validity and extract the priority level embedded in the token. We compute the effective priority level $eprio$ from its priority level $prio$ and the request rate $r$ from the client. We have implemented a simple weighted probabilistic fair queuing filter to rate limit requests from a client using its effective priority level ($eprio$).

**Server Side Application Layer.** We use Apache Tomcat filters to hook on HTTP request processing before an incoming request is forwarded to the servlet engine. This filter is used to record the time at which request processing starts. Similarly, a filter on an outgoing response is used to record the time at which the request processing ended. This filter provides the application programmers 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.

*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.

**Challenge Server.** We have implemented an adaptive challenge mechanism that is similar to the one described in [40]. Client side implementation of the challenge solver is implemented using 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 the challenge server can generate about one million challenges per second and check about one million challenges per second. The challenge server can generate up to one million trust tokens per second. This rate is primarily limited by the cost of computing a message authentication code (MAC) on a 16 Byte input using HMAC-SHA1 (0.91$\mu$s). Given that the challenge server can handle very high request rates and it serves only two types of requests (challenge generation and solution verification) it would be very hard for an adversary to launch application level DoS attacks 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.

Our experiments showed that a client side challenge solver using a C program, a Java applet and JavaScript requires 1 second, 1.1 seconds and 1012 seconds (respectively) to solve a challenge with hardness $m$=20. A JavaScript based challenge solver is unfair to the legitimate clients since the attackers can use any mechanism (including a non-client transparent C program) to solve the challenge. Therefore, we chose to adopt the client transparent Java applet based challenge solver whose performance is comparable to that of a C program based challenge solver.

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

Table 1: Overhead

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

Table 2: 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 3: Attack Strategies

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

Table 4: Attack Types

| | |
|---|---|
| A1 | Apache HTTPD |
| A2 | TPCW |

Table 5: Applications

## 4 Evaluation

In this section, we present two sets of experiments. The first set of experiments quantifies the overhead of our trust token filter. The second set of experiments demonstrates the effectiveness of our approach against application level DoS attacks.

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 services in our experiments. The first service is a bandwidth intensive Apache HTTPD service [2]. 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 Fire-Fox [14] running on Linux. The web browser was instrumented to programmatically

send requests to the server using JavaScripts [29]. We measured the average client side throughput in web pages per second (WPPs) as the performance metric. 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, amply demonstrating the portability of our approach.

The second service is a database intensive web transaction processing benchmark TPCW 1.0 [37]. We used a Java based workload generator from PHARM [31]. We used Apache Tomcat 5.5 [3] as our web server and IBM DB2 8.1 [20] 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.

We simulated two types of clients: one good client and up to a hundred DoS attackers connected via a 100 Mbps LAN to the server. The firewall functionality described in Section 3.3 is implemented on the server. The good client was used to measure the throughput of the web server under a DoS attack. The intensity of the 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. In the following experiments we show that under application level DoS attacks, the breakdown point for the trust token filter ($tt$) is much larger than that for other state of the art DoS filters.
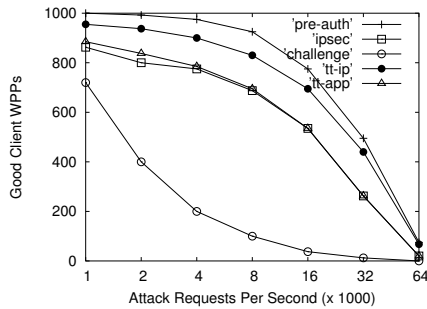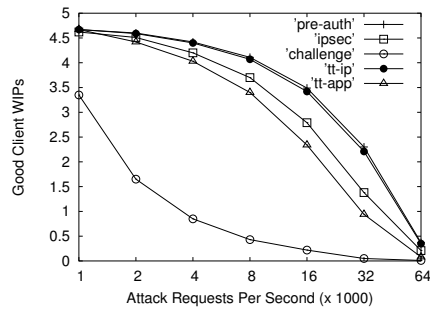


Fig. 4: $\langle S1, T1, A1 \rangle$        Fig. 5: $\langle S1, T1, A2 \rangle$

## 4.1  Performance Overhead

Table 1 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,
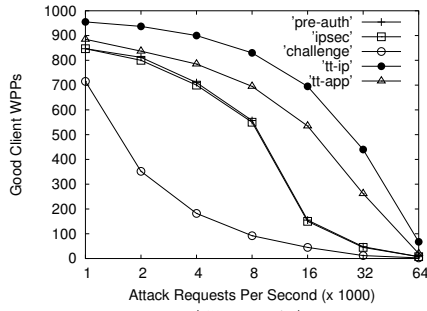
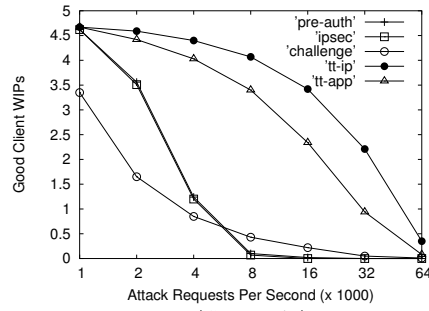Fig. 6: $\langle S1, T2, A1 \rangle$



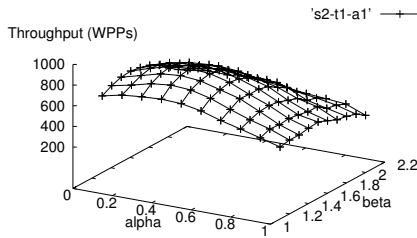Fig. 7: $\langle S1, T2, A2 \rangle$
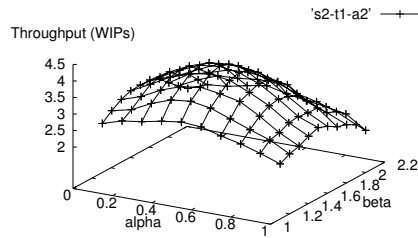


Fig. 8: $\langle S2, T1, A1 \rangle$



Fig. 9: $\langle S2, T1, A2 \rangle$

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 1 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 1 shows that the overhead of these two implementations are comparable; however, in section 4.2 we show that 'tt-ip' offers better resilience to DoS attacks.

### 4.2 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 3), attack type $T$ (Table 4) and application $A$ (Table 5). 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 2). 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 [44]. 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 4 and 5 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 [33][45] and ingress filtering [13].

**Trust token filter is resilient to application level DoS attacks**: $\langle S1, T2, A1 \rangle$ and $\langle S1, T2, A2 \rangle$. Table 2 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 6 and 7 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 6 and 7 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 6 and 7 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 4, 5, 6 and 7 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 is 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.

### 4.3 Attacks on Trust Token Filter

In Section 4.2 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 $\alpha$ and $\beta$:** $\langle S2, T1, A1 \rangle$ and $\langle S2, T1, A2 \rangle$. Figures 8 and 9 show the throughput for a good client for various values of $\alpha$ and $\beta$ using applications HTTPD and TPCW respectively. Recall that $\alpha$ and $\beta$ are the parameters used for the additive increase and multiplicative decrease policy for updating a client's priority level (see Section 3). 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 $\alpha$ and $\beta$ 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 6 and 7).

Figure 10 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 $\alpha$ and $\beta$ maximize the throughput for a good client. Note that a large $\alpha$ boosts the priority level for good clients while a large $\beta$ penalizes the bad clients heavily. This suggests that one may dynamically vary the values of $\alpha$ and $\beta$ depending on the server load.

**Server resource utilization parameter $\gamma$:** $\langle S2, T2, A1 \rangle$ and $\langle S2, T2, A2 \rangle$. Figures 11 and 12 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 $\gamma$. Recall that we use the parameter $\gamma$ to weigh a request's response time against the request's utility (see Section 3). If $\gamma$ 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 $\gamma$ 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 2). The figures show that one can obtain optimal values for the filter parameter $\gamma$ that maximizes the average throughput for a good client. The optimal value for parameter $\gamma$ ensures that the drop in throughput
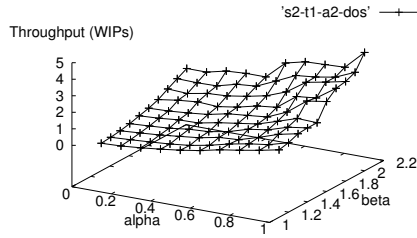
Fig. 10: $\langle S2, T1, A2\rangle$
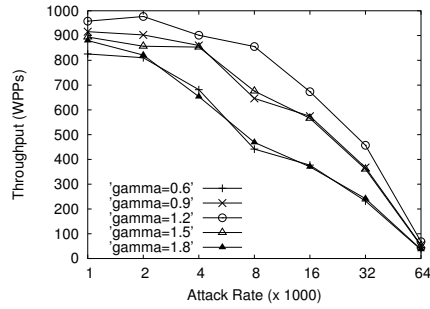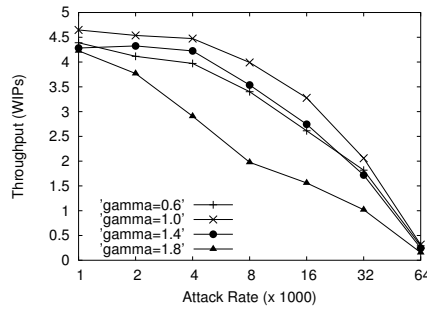


Fig. 11: $\langle S2, T2, A1\rangle$
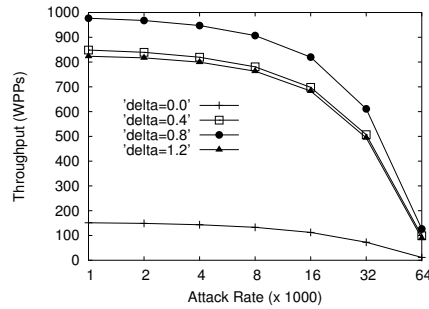


Fig. 12: $\langle S2, T2, A2\rangle$



Fig. 13: $\langle S2, T3, A1\rangle$

for HTTPD and TPCW is 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
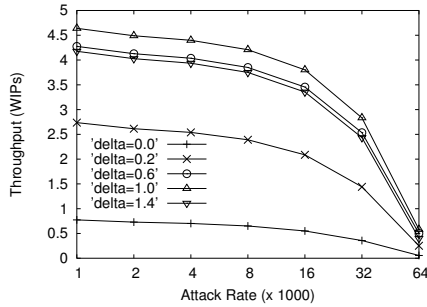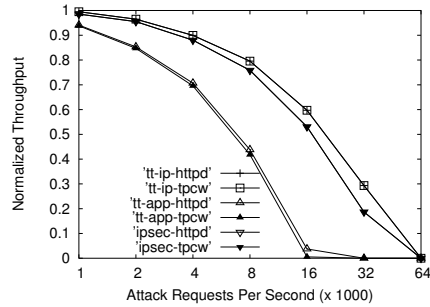


Fig. 14: $\langle S2, T3, A2\rangle$



Fig. 15: $\langle S1, T4, A1\rangle$ and $\langle S1, T4, A2\rangle$

$\langle S2, T3, A2\rangle$. Figures 13 and 14 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 $\delta$, which is used to penalize (possibly) old trust tokens. A small value of $\delta$ permits attackers to use older tokens while a large value of $\delta$ may result in rejecting requests even from well behaving clients. The figures show that one can obtain optimal values for the filter parameter $\delta$ that maximize the average throughput for a good client. Using the optimal value for parameter $\delta$ 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 15 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 1 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').

## 5 Discussion

### 5.1 Limitations and Open Issues

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 [12] 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. Hence, one can modify the trust token as: $tt = \langle prio, tv, H_{MK}(cip, cpn, sip, tv, prio) \rangle$, where $cip$ denotes the IP address of the NAT router and $cpn$ refers to the client's translated port number as assigned by the NAT router.

However, HTTP proxies do not operate using port address translation (PAT). One potential solution is to deploy a trust token filter at the HTTP proxy. The trust token filter at a HTTP proxy gets application specific priority updates for a client's request from the web server. While the web server may not know the set of requests that originated from one client, the HTTP proxy can aggregate priority updates of all requests on a per-client basis. It can use this per-client priority information to filter future HTTP requests from its clients. 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 application level DoS attacks is an open problem.

## 5.2   Related Work

Several past papers have addressed network level DoS attacks [4][33][13][18][45]. These techniques are useful in defending a server against network level bandwidth exhaustion attacks. However, the lack of application semantics and domain knowledge render network level DoS filters incapable of handling application level DoS attacks. Several tools have been proposed to perform preauthorization based DoS protection [25][30][26]. Our experiments show that even if preauthorization based techniques were feasible, an application level DoS attack by a small fraction of malicious preauthorized clients can jeopardize the system. Several authors have proposed challenge based mechanisms for DoS protection [24][40][22][36][41][39][11]. 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 [10] present DoS attacks that target application level hash tables by introducing collisions. Section 2 provides a more detailed discussion on the above mentioned DoS protection mechanisms.

Recently, several web applications (including Google Maps [17] and Google Mail [16]) have adopted the Asynchronous JavaScript and XML (AJAX) model [42]. 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 handling HTTP cookies and solving cryptographic challenges. Recent surveys indicate that at least 97% of the client browsers support JavaScript and Java [43][38].

Jung et al. [23] 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. An adversary can thwart IP address based clustering by employing a DDoS attack wherein the zombie machines are uniformly distributed over several IP domains. File reference characteristics may not be sufficient to mitigate application level DoS attacks since the cost of serving a request may be a complex function of the parameters embedded in the request. Siris et al. [34] 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.

Several papers have presented techniques for implementing different QoS guarantees for serving web data [7][9][5]. A summary of past work in this area is provided in [21]. These papers are not targeted at preventing DoS attacks and do not discuss application level DoS attacks.

## 6   Conclusion

In this paper we have proposed a middleware to protect a website against application level DoS attacks. We have developed a trust token filter that allocates more resources to the good clients, while severely restricting the amount of resources allocated to the

DoS attackers. Our approach works by adaptively setting a client's priority level in response to the client's requests, in a way that incorporates application level semantics. Our DoS protection mechanism is proactive, client transparent, and capable of mitigating application level DoS attacks that may not be known a priori. We have described a concrete implementation of our proposal on the Linux kernel and presented a detailed evaluation using two workloads: a bandwidth intensive Apache HTTPD benchmark and TPCW (running on Apache Tomcat and IBM DB2). Our experiments demonstrate the advantages of the trust token filter over other network level DoS filters in defending against application level DoS attacks.

# References

[1] Netfilter/IPTables project homepage. http://www.netfilter.org/.
[2] Apache. Apache HTTP server. http://httpd.apache.org.
[3] Apache. Apache tomcat servlet/JSP container. http://jakarta.apache.org/tomcat.
[4] D. J. Bernstein. SYN cookies. http://cr.yp.to/syncookies.html, 2005.
[5] V. Cardellini, E. Casalicchio, M. Colajanni, and M. Mambelli. Enhancing a web server cluster with quality of service mechanisms. In *Proceedings of 21st IEEE IPCCC*, 2002.
[6] CERT. Incident note IN-2004-01 W32/Novarg.A virus, 2004.
[7] S. Chandra, C. S. Ellis, and A. Vahdat. Application-level differentiated multimedia web services using quality aware transcoding. In *Proceedings of IEEE special issue on QoS in the Internet*, 2000.
[8] H. Chen and A. Iyengar. A tiered system for serving differentiated content. In *Proceedings of World Wide Web: Internet and Web Information Systems Vol. 6, No. 4*, December 2003.
[9] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for web QoS. In *Proceedings of IEEE Transactions on Computers*, 2002.
[10] S. A. Crosby and D. S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of 12th USENIX Security Symposium, pp: 29-44*, 2003.
[11] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *Proceedings of Crypto*, 1992.
[12] K. Egevang and P. Francis. RFC 1631: The IP network address translator (NAT). http://www.faqs.org/rfcs/rfc1631.html, 1994.
[13] R. Ferguson and D. Senie. RFC 2267: Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. http://www.faqs.org/rfcs/rfc2267.html, 1998.
[14] FireFox. Mozilla firefox web browser. http://www.mozilla.org/products/firefox, 2005.
[15] A. fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. gauthier. Cluster-based scalable network services. In *Proceedings of 16th ACM SOSP*, 1997.
[16] Google. Google mail. http://mail.google.com/.
[17] Google. Google maps. http://maps.google.com/.
[18] Halfbakery. Stateless TCP/IP server. http://www.halfbakery.com/idea/Stateless_20TCP_2fIP_20server.
[19] IBM. IBM network dispatcher features. http://www-3.ibm.com/software/network/about/features/keyfeatures.html.
[20] IBM. DB2 universal database. http://www-306.ibm.com/software/data/db2, 2005.

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

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

[23] J. Jung, B. Krishnamurthy, and M. rabinovich. Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites. In *Proceedings of 10th WWW Conference*, 2002.

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

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

[26] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure overlay services. In *Proceedings of the ACM SIGCOMM*, 2002.

[27] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. http://www.faqs.org/rfcs/rfc2104.html, 1997.

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

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

[30] OpenSSL. Openssl. http://www.openssl.org/.

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

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

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

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

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

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

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

[38] W3Schools. Browser statistics. http://www.w3schools.com/browsers/browsers_stats.asp.

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

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

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

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

[43] Wikipedia. Comparison of web browsers. http://en.wikipedia.org/wiki/Comparison_of_web_browsers.

[44] B. Yang and H. Garcia-Molina. Improving search in peer-to-peer networks. In *Proceedings of 22nd IEEE ICDCS*, 2002.

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