DDoS Defense by Offense

Michael Walfish*, Mythili Vutukuru*, Hari Balakrishnan*, David Karger*, and Scott Shenker†

*MIT, {mwalfish,mythili,hari,karger}@csail.mit.edu †UC Berkeley and ICSI, shenker@icsi.berkeley.edu

ABSTRACT

This paper presents the design, implementation, analysis, and experimental evaluation of *speak-up*, a defense against *application-level* distributed denial-of-service (DDoS), in which attackers cripple a server by sending legitimate-looking requests that consume computational resources (*e.g.*, CPU cycles, disk). With speak-up, a victimized server encourages all clients, resources permitting, *to automatically send higher volumes of traffic*. We suppose that attackers are already using most of their upload bandwidth so cannot react to the encouragement. Good clients, however, have spare upload bandwidth and will react to the encouragement with drastically higher volumes of traffic. The intended outcome of this traffic inflation is that the good clients crowd out the bad ones, thereby capturing a much larger fraction of the server's resources than before. We experiment under various conditions and find that speak-up causes the server to spend resources on a group of clients in rough proportion to their aggregate upload bandwidth. This result makes the defense viable and effective for a class of real attacks.

Categories and Subject Descriptors: C.2.0 [Computer-Communication Networks]: Security and protection

General Terms: Design, Experimentation, Security

Keywords: DoS attack, bandwidth, currency

1 INTRODUCTION

Our goal is to defend servers against *application-level* Distributed Denial of Service (DDoS), a particularly noxious attack in which computer criminals mimic legitimate client behavior by sending proper-*looking* requests via compromised and commandeered hosts [10, 18, 36, 37]. By exploiting the fact that many Internet servers have "open clientele" (*i.e.*, they cannot tell a *good* client from the request alone), the attacker forces the victim server to spend much of its resources on spurious requests. For the savvy attacker, the appeal of this attack over a classic ICMP link flood is two-fold. First, it requires far less bandwidth: the victim's computational resources—disks, CPUs, memory, application server licenses, etc.—can often be depleted by proper-looking requests long before its access link is saturated. Second, because the attack traffic is "in-band", it is harder to identify and thus more potent. Examples of such (often extortionist [30, 44]) attacks include using bots to attack Web sites by: requesting large files [36, 37], making queries of search engines [10], and issuing computationally expensive requests (*e.g.*, database queries or transactions) [21].

Current DDoS defenses try to *slow down the bad clients*. Though we stand in solidarity with these defenses in the goal of limiting the service that attackers get, our approach is different. We rely on *encouragement* (a term made precise in §3), whereby the server causes a client, resources permitting, to automatically send a higher volume of traffic. Our approach is to encourage *all clients to speak up*, rather than sit idly by while attackers drown them out. For if, as we suppose, bad clients are already using most of their upload bandwidth, then encouragement will not change their traffic volume. However, the good clients typically use only a small fraction of their available bandwidth to send requests, so they will react to encouragement by drastically increasing their traffic volume. As good clients send more traffic, the traffic into the server inflates, but the good clients will be much better represented in the mix and thereby capture a much larger portion of the server than before.

Of course, this caricature of our approach leaves many mechanisms unmentioned and myriad issues unaddressed. The purpose of this paper is to bring the preceding high-level description to life with a viable and effective system. To that end, we describe the design, prototype implementation, and evaluation of *speak-up*, a defense against application-level DDoS attacks in which clients are encouraged to send more traffic to an attacked server.

We put our approach in context with the following taxonomy of defenses:

Over-provision massively. In theory, one could purchase enough computational resources to serve attackers and good clients. However, anecdotal evidence suggests that while sites provision additional *link* capacity during attacks [33], even the largest Web sites try to conserve *computation* by detecting and denying access to bots [30, 42] using the methods in the next category.

Detect and block. These approaches try to distinguish between good and bad clients. Examples are profiling by IP address [5, 9, 27] (a box in front of the server or the server itself admits requests according to a learned demand profile); rate-limiting alone (a special case of profiling in which the acceptable request rate is the same for all clients); CAPTCHA-based defenses [16, 21, 29, 42, 47] that preferentially admit humans; and capabilities [4, 50, 51] (the network allows only traffic that the recipient has authorized). These techniques are powerful because they seek to block or explicitly limit unauthorized users, but their discriminations can err (see §8.1). Moreover, they cannot easily handle *heterogeneous* requests (*i.e.*, those that cause the server to do different amounts of work). The next category addresses these limitations.

Charge all clients in a currency. Here, an attacked server gives a client service only after it pays in some currency. Examples are CPU or memory cycles (evidence of payment is the solution to a computational puzzle) [1, 6, 7, 11, 12, 20, 25, 49] and money [25]. With these defenses, there is no need to discriminate between good and bad clients, and the server can require a client to pay more for "hard" requests. However, for the legitimate users to capture the bulk of the service, they must in aggregate have more of the currency than the attackers.

In this taxonomy, speak-up is a currency approach with *bandwidth as the currency*. We believe that this work is the first to investigate

SIGCOMM '06, September 11–15, 2006, Pisa, Italy.

this idea (though it was proposed in a workshop paper by us [48] and [17, 39] share the same high-level motivation; see §8.1).

The central mechanism in speak-up is a server front-end, the *thinner*, that protects the server from overload and performs encouragement (§3). Encouragement can take several forms (§3.2, §3.3). The one that we implement and evaluate is a *virtual auction*: when the server is overloaded, the thinner causes each new client to automatically send a congestion-controlled stream of dummy bytes on a separate payment channel, and when the server is ready to process a request, the thinner selects the client that has sent the most bytes (§3.3). We show that the ability to "game" this scheme is limited (§3.4). We also design an extension of the thinner to handle heterogeneous requests (§5).

As a concrete instantiation of speak-up, we implemented the thinner as a Web front-end (§6). The thinner performs encouragement by giving JavaScript to unmodified Web clients that makes them send large HTTP POSTs. These POSTs are the "bandwidth payment". We find that this implementation meets our goal of allocating the protected server's resources in rough proportion to clients' upload bandwidth (§7). Despite being unoptimized, the implementation sinks 1.5 Gbits/s on a high-end PC.

Practical DDoS mitigation requires multiple techniques, and speak-up is not intended to stand alone. In §8, we compare speak-up to other defenses and discuss when it should work with them.

2 APPLICABILITY OF SPEAK-UP

Before describing speak-up's design, we discuss under what conditions and to what extent speak-up is useful. We start by informally addressing four commonly asked questions and then characterize our threat model and speak-up's range of applicability.

2.1 Four Questions

How much aggregate bandwidth does the legitimate clientele need for speak-up to be effective? Speak-up helps good clients, no matter how much bandwidth they have. Speak-up either ensures that the good clients get all the service they need or increases the service they get (compared to an attack without speak-up) by the ratio of their available bandwidth to their current usage, which we expect to be very high. Moreover, as with many security measures, speak-up "raises the bar" for attackers: to inflict the same level of service-denial on a speak-up defended site, a much larger botnet—perhaps several orders of magnitude larger—will be required. Similarly, the amount of over-provisioning needed at a site defended by speak-up is much less than what a non-defended site would need.

Thanks for the sales pitch, but what we meant was: how much aggregate bandwidth does the legitimate clientele need for speak-up to leave them unharmed by an attack? The answer depends on the server's spare capacity (*i.e.*, 1−utilization) when unattacked. Speak-up's goal is to allocate resources in proportion to the bandwidths of requesting clients. If this goal is met, then for a server with spare capacity 50%, the legitimate clients can retain full service if they have the same aggregate bandwidth as the attacking clients (see §3.1). For a server with spare capacity 90%, the legitimate clientele needs only 1/9th of the aggregate bandwidth of the attacking clients.

We now put these results in the context of today's botnets by first noting that most botnets today are less than 100,000 hosts, and even 10,000 hosts is a large botnet [18, 19]. (Supporting evidence for these sizes is as follows. One study found that the average bot has roughly 100 Kbits/s of bandwidth [40]. If each bot uses half its bandwidth during an attack, then a 10,000-node botnet generates 500 Mbits/s of traffic, and a 100,000-node botnet generates 5 Gbits/s of traffic. These numbers are above, respectively, the 80th percentile and 99th percentile of attack sizes observed in [38].) Second, assume that the average good client also has 100 Kbits/s of bandwidth. Then for a service whose spare capacity is 90%, speak-up can fully defend it (*i.e.*, leave its good clients unharmed) against a 10,000-host (resp., 100,000-host) botnet if the good clients number ~1,000 (resp., ~10,000).

We believe that these orders of magnitude are not larger than the clientele of the Web's largest sites: these numbers refer to the good clients currently *interested* in the service, many of which may be quiescent. For example, consider search engines. Humans pausing between queries count in the "current clientele", and there are almost certainly thousands of such users at any time for the large search engines.

Then couldn't small Web sites, even if defended by speak-up, still be harmed? Yes. For botnets of the sizes just mentioned (and for the small number of even larger ones [18, 19, 43]), speak-up-defended sites need a large clientele or vast over-provisioning to fully withstand attack. However, we think that future botnets will be smaller.

Our rationale is as follows. Today, sites can recognize primitive bots. Such bots launch attacks too quickly, and sites block them by profiling IP addresses. To evade these defenses, bots will eventually become more sophisticated, for example by building up an activity profile at a given Web site and then flying under the profiling radar during an attack. At this point, it will be hard for *sites* to identify and block the bots. However, *ISPs*, which can observe their hosts over long periods of time, will still be able to identify bots. Indeed, we speculate that once sites no longer have effective defenses, society (governments, public and industry pressure, etc.) will force ISPs to act, thereby reducing the number of bots (but not eliminating them—bot identification is not a precise science). When attackers adapt to having fewer but smarter bots, application-level attacks—which require smart bots but conserve resources—will be more common, making speak-up more broadly applicable.

Because bandwidth is in part a communal resource, doesn't the encouragement to send more traffic damage the network? We first observe that speak-up inflates traffic only to servers currently under attack—a very small fraction of all servers—so the increase in total traffic will be minimal. Moreover, the "core" appears to be heavily over-provisioned (see, *e.g.*, [15]), so it could absorb such an increase. Finally, speak-up's additional traffic is congestion-controlled and will share fairly with other traffic. We address this question more fully in §4 and other issues raised by speak-up in §9.

2.2 Threat Model and Applicability Conditions

The preceding informal discussion gave a general picture of speak-up's applicability. We now give a more precise description, beginning with the threat model. Speak-up aims to protect a *server*, defined as any network-accessible service with scarce computational resources (disks, CPUs, RAM, application licenses, file descriptors, etc.), from an *attacker*, defined as an entity (human or organization) that is trying to deplete those resources with legitimate-looking requests (database queries, HTTP requests, etc.). Such an assault is called an *application-level attack* [18].

Each attacker sends traffic from many compromised hosts, and this traffic obeys all protocols, so the server has no easy way to tell from a single request that it was issued with ill intent. Most services handle requests of varying difficulty (*e.g.*, database queries with very different completion times). While servers may not be able to determine a request's difficulty *a priori*, our threat model presumes that the attacker *can* send difficult requests intentionally.

One reason that application-level attacks are challenging to thwart is that the Internet has *no robust notion of host identity*. For datagram protocols without three-way handshakes (*e.g.*, DNS-over-UDP), spoofing is trivial, and even for protocols with three-way handshakes, spoofing is possible. (Such spurious handshakes—observed before [41] and correlated with spam transmissions [34]—work because many ISPs accept spurious BGP routes and propagate them to other ISPs [14].) Since a determined attacker can repeatedly request service from a site while pretending to have different IP addresses, we assume that an abusively heavy client of a site will not always be identifiable as such.

We are not considering link attacks. We assume that the server's access links (and, more generally, the network infrastructure) are not flooded; see condition C1 below.

There are many types of Internet services, with varying defensive requirements; speak-up is not appropriate for all of them. For speak-up to defend against the threat modeled above, the following two conditions must hold:

C1 **Adequate link bandwidth.** The protected service needs enough link bandwidth to handle the incoming request stream (and this stream will be inflated by speak-up). A server can satisfy this condition via a high-bandwidth access link or co-location at a data center. However, we expect the more common deployment to be ISPs—which of course have significant bandwidth—offering speak-up as a service (just as they do with other DDoS defenses today), perhaps amortizing the expense over many defended sites, as suggested in [2].

C2 **Adequate client bandwidth.** To be unharmed during an attack, the good clients must have in total roughly the same order of magnitude (or more) bandwidth than the attacking clients. As argued in §2.1, this property holds for some sites today, and we expect it to hold for many more in the future.

Furthermore, speak-up offers advantages over alternate defenses when all of the following also hold:

C3 **No pre-defined clientele.** Otherwise, the server can install filters or use capabilities [4, 50, 51] to permit only traffic from known clients.

C4 **Non-human clientele.** If the clientele is exclusively human, one may be able to use proof-of-humanity tests (*e.g.*, [16, 21, 29, 31, 42, 47]).

C5 **Unequal requests *or* spoofing *or* smart bots.** If the server has an unequal request load (as mentioned before), then our currency-based approach can charge clients for harder requests—even if the server does not know the request difficulty *a priori* (see §5). Also, if attackers spoof rampantly (as mentioned above), traditional defenses based on identifying and blocking clients are unlikely to keep the bots at bay. Likewise, those defenses could be confounded by bots smart enough to fly under the profiling radar (as discussed in §2.1).

The canonical example of a service that meets all of the conditions above (provided its clientele has adequate bandwidth) is a Web server for which requests are computationally intensive, perhaps because they involve back-end database transactions or searches (*e.g.*, sites with search engines, travel sites, and automatic update services for desktop software). Often, the clientele of these sites is partially or all non-human. Beyond these server applications, speak-up could protect the capability allocator in network architectures such as TVA [51] and SIFF [50] that seek to handle DoS attacks by issuing capabilities to clients.

3 DESIGN OF SPEAK-UP

Speak-up is motivated by a simple observation about bad clients: they send requests to victimized servers at much higher rates than legitimate clients do. (This observation has also been made by many others, including the authors of profiling and detection methods.) At the same time, *some* limiting factor must prevent bad clients from sending even more requests. We posit that in many cases this limiting factor is bandwidth. The specific constraint could be a physical limit (*e.g.*, access link capacity) or a threshold above which the attacker fears detection by profiling tools at the server or by the human owner of the "botted" host. For now, we assume that bad clients exhaust all of their available bandwidth on spurious requests. In contrast, good clients, which spend substantial time quiescent, are likely using a only small portion of their available bandwidth. The key idea of speak-up is to exploit this difference, as we now explain with a simple illustration.

Illustration. Imagine a request-response server, where each request is cheap for clients to issue, is expensive to serve, and consumes the same quantity of server resources. Real-world examples include single-packet Web requests, DNS front-ends (*e.g.*, those used by content distribution networks or infrastructures like CoDoNS [35]), and AFS servers. Suppose that the server has the capacity to handle c requests per second and that the aggregate demand from good clients is g requests per second, $g < c$. Assume that when the server is overloaded it randomly drops excess requests. If the attackers consume all of their aggregate upload bandwidth, B (which for now we express in requests per second) in attacking the server, and if $g+B > c$, then the good clients will receive only a fraction $\frac{g}{g+B}$ of the server's resources. Assuming $B \gg g$ (if $B \approx g$, then over-provisioning by moderately increasing c would ensure $g + B < c$, thereby handling the attack), the bulk of the server goes to the attacking clients. This situation is depicted in Figure 1(a).

In this situation, current defenses would try to slow down the bad clients. But what if, instead, we arranged things so that when the server is under attack *good clients send requests at the same rates as bad clients*? Of course, the server does not know which clients are good, but the bad clients have already "maxed out" their bandwidth (as assumed above). So if the server encouraged *all* clients to use up their bandwidth, it could speed up the good ones without telling apart good and bad. Doing so would certainly inflate the traffic into the server during an attack. But it would also cause the good clients to be much better represented in the mix of traffic, giving them much more of the server's attention and the attackers much less. If the good clients have total bandwidth G, they would now capture a fraction $\frac{G}{G+B}$ of the server's resources, as depicted in Figure 1(b). Since $G \gg g$, this fraction is much larger than before.

We now focus on speak-up's design, which aims to make the preceding under-specified illustration practical. In the rest of this section, we assume that all requests cause equal server work. We begin with requirements (§3.1) and then develop two ways to realize these requirements (§3.2, §3.3). We also consider various attacks (§3.4). We revisit our assumptions in §4 and describe how speak-up handles heterogeneous requests in §5.

3.1 Design Goal and Required Mechanisms

Design Goal. In keeping with our view of bandwidth as a currency, our principal goal is to allocate resources to competing clients in proportion to their bandwidths:[1]

[1]This goal might seem more modest than the chief aim of profiling: blocking bad clients altogether. However, as discussed in §8.1, given a smart bot, profiling can only limit, not block, bad clients.

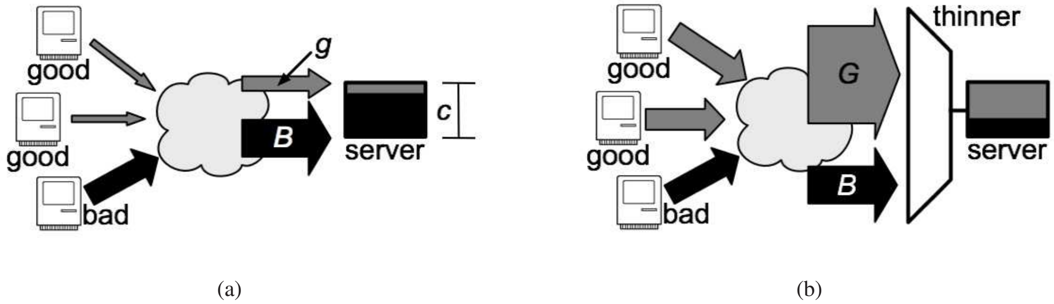

Figure 1: An attacked server, $B + g > c$, (a) without speak-up (b) with speak-up. The good clients' traffic is gray, as is the portion of the server that they capture. The figure does not specify speak-up's encouragement mechanism (aggressive retries or payment channel).

> If the good clients make g requests per second in aggregate and have an aggregate bandwidth of G requests per second to the server, and if the bad clients have an aggregate bandwidth of B requests per second, then the server should process good requests at a rate of $\min(g, \frac{G}{G+B}c)$ requests per second.

If this goal is met, then modest over-provisioning of the server (relative to the legitimate demand) can satisfy the good clients. For if it is met, then satisfying them requires only $\frac{G}{G+B}c \geq g$ (*i.e.*, the piece the good clients *can* get must exceed their demand). This expression translates to the *idealized server provisioning requirement*:

$$c \geq g(1 + B/G) \stackrel{\text{def}}{=} c_{id},$$

which says that the server must be able to handle the "good" demand (g) and diminished demand from the bad clients ($B\frac{g}{G}$). For example, if $B = G$ (a special case of condition C2 in §2.2), then the required over-provisioning is a factor of two ($c \geq 2g$). In practice, speak-up cannot exactly achieve this ideal because limited cheating is possible. We analyze this effect in §3.4.

Required Mechanisms. Any practical realization of speak-up needs three mechanisms. The first is a way to limit requests to the server to c per second. However, rate-limiting alone will not change the server's allocation to good and bad clients. Since the design goal is that this allocation reflect available bandwidth, speak-up also needs a mechanism to reveal that bandwidth: speak-up must perform *encouragement,* which we define as *causing a client to send more traffic—potentially much more—for a single request than it would if the server were unattacked*. Third, given the incoming bandwidths, speak-up needs a *proportional allocation* mechanism to admit clients at rates proportional to their delivered bandwidth.

To implement these mechanisms, speak-up uses a front-end to the server, called the *thinner*, depicted in Figure 1(b). The thinner implements encouragement and controls which requests the server sees. Encouragement can take several forms; the two variations of speak-up below, in §3.2 and §3.3, each incorporate a different one with correspondingly distinct proportional allocation mechanisms. Before presenting these, we observe that today when a server is overloaded and fails to respond to a request, a client typically times out and retries—thereby generating more traffic than if the server were unloaded. However, the bandwidth increase is small (since today's timeouts are long). In contrast, encouragement (which is initiated by an agent of the *server*) will cause good clients to send significantly more traffic—while still obeying congestion control.

3.2 Random Drops and Aggressive Retries

In the version of speak-up that we now describe, the thinner implements proportional allocation by dropping requests at random to reduce the rate to c. To implement encouragement, the thinner, for each request that it drops, immediately asks the client to retry. This synchronous *please-retry* signal causes the *good* clients—the bad ones are already "maxed out"—to retry at far higher rates than they would under silent dropping. (Silent dropping happens in many applications and in effect says, "please try again later", whereas the thinner says, "please try again *now*".)

With the scheme as presented thus far, a good client sends only one packet per round-trip time (RTT) while a bad client can keep many requests outstanding, thereby manufacturing an advantage. To avoid this problem, we modify the scheme as follows: without waiting for explicit please-retry signals, the clients send repeated retries in a congestion-controlled stream. Here, the feedback used by the congestion control protocol functions as implicit please-retry signals. This modification allows all clients to pipeline their requests and keep their pipe to the thinner full.

One might ask, "To solve the same problem, why not enforce one outstanding retry per client?" or, "Why not dispense with retries, queue clients' requests, and serve the oldest?" The answer is "spoofing and NAT". Spoofing, as happens in our threat model (§2.2), means that one client may claim to be several, and NAT means that several clients (which may individually have plenty of bandwidth) may appear to be one. Thus, the thinner can enforce neither one outstanding retry per "client" nor any other quota scheme that needs to identify clients. Ironically, taxing clients is easier than identifying them: the continuous stream of bytes that clients are asked to send ensures that each is charged individually.

Indeed, speak-up is a currency-based scheme (as we said earlier), and the price for access is the number of retries, r, that a client must send. Observe that the thinner does not communicate r to clients: good clients keep resending until they get through (or give up). Also, r automatically changes with the attack size.

This approach fulfills the design goal in §3.1, as we now show. The thinner admits incoming requests with some probability p to make the total load reaching the server be c. There are two cases. Either the good clients cannot afford the price, in which case they exhaust all of their bandwidth and do not get service at rate g, or they can afford the price, in which case they send retries until getting through. In both cases, the price, r, is $1/p$. In the first case, a load of $B + G$ enters the thinner, so $p = \frac{c}{B+G}$, $r = \frac{B+G}{c}$, and the good clients can pay for $G/r = \frac{G}{G+B}c$ requests per second. In the second case, the good clients get service at rate g, as required.

3.3 Explicit Payment Channel

We now describe another encouragement mechanism, which we use in our implementation and evaluation. Conceptually, the thinner asks clients to pad their requests with dummy bytes. However, instead of having to know the correct amount of padding and communicate it to clients, the thinner does the following. When the server is overloaded, the thinner asks a requesting client to open a separate *payment channel*. The client then sends a congestion-controlled stream of bytes on this channel. We call a client that is sending bytes a *contending* client; the thinner tracks how many bytes each contending client sends. Assume that the server notifies the thinner when it is ready for a new request. When the thinner receives such a notification, it holds a *virtual auction*: it admits to the server the contending client that has sent the most bytes, and it terminates the corresponding payment channel.

As with the version in §3.2, the price here emerges naturally. Here, it is expressed in bytes per request. The "going rate" for access is the winning bid from the most recent auction. We now consider the average price. Here, we express B and G in bytes (not requests) per second and assume that the good and bad clients are "spending everything", so $B+G$ bytes per second enter the thinner. Since auctions happen every $1/c$ seconds on average, the average price is $\frac{B+G}{c}$ bytes per request.

However, we cannot claim, as in §3.2, that good clients get $\frac{G}{G+B}c$ requests served per second: the auction might allow "gaming" in which adversaries consistently pay a lower-than-average price, forcing good clients to pay a higher-than-average price. We show in §3.4 that the auction can be gamed but not too badly, so all clients do in fact see prices that are close to the average.

Comparison. There are two main differences between the scheme in §3.2 and this one. First, with the other scheme, the thinner must determine p and apply it in a way that cannot be "gamed"; here, the thinner's rule is simply to select the top-paying client. Second, with the other scheme, clients pay in-band. Which option is appropriate—payment in-band or on a separate channel—depends on the application. For example, our prototype (§6) needs the latter option for reasons related to how JavaScript drives Web browsers.

3.4 Robustness to Cheating

In considering the robustness of the virtual auction mechanism, we begin with a theorem and then describe how practice may be both worse and better than this theory. The theorem is based on one simplifying assumption: that requests are served with perfect regularity (*i.e.*, every $1/c$ seconds).

Theorem 3.1 *In a system with regular service intervals, any client that continuously transmits an ϵ fraction of the average bandwidth received by the thinner gets at least an $\epsilon/2$ fraction of the service, regardless of how the bad clients time or divide up their bandwidth.*

Proof: Consider a client, X, that transmits an ϵ fraction of the average bandwidth. The intuition is that to keep X from winning auctions, the other clients must deliver substantial payment.

Because our claims are purely about proportions, we choose units to keep the discussion simple. We call the amount of bandwidth that X delivers between every pair of auctions a *dollar*. Suppose that X must wait t auctions before winning k auctions. Let t_1 be the number of auctions that occur until (and including) X's first win, t_2 the number that occur after that until and including X's second win, and so on. Thus, $\sum_{i=1}^{k} t_i = t$. Since X does not win until auction number t_1, X is defeated in the previous auctions. In the first auction, X has delivered 1 dollar, so at least 1 dollar is spent to defeat it; in the next auction 2 dollars are needed to defeat it, and so on until the $(t_1-1)^{st}$ auction when t_1-1 dollars are spent to defeat it. So $1+2+\cdots+(t_1-1) = t_1(t_1-1)/2$ dollars are spent to defeat X before it wins. More generally, the total dollars spent by other clients over the t auctions is at least $\sum_{i=1}^{k} \frac{t_i^2-t_i}{2} = \sum_{i=1}^{k} \frac{t_i^2}{2} - \frac{t}{2}$. This sum is minimized, subject to $\sum t_i = t$, when all the t_i are equal, namely $t_i = t/k$. We conclude that the total spent by the other clients is at least $\sum_{i=1}^{k} \frac{t^2}{2k^2} - \frac{t}{2} = \frac{t^2}{2k} - \frac{t}{2}$.

Adding the t dollars spent by X, the total number of dollars spent is at least $\frac{t^2}{2k} + \frac{t}{2}$. Thus the *fraction* of the total spent by X, which we called ϵ, is at most $2/(t/k+1)$. It follows that $k/t \geq \frac{\epsilon}{2-\epsilon} \geq \epsilon/2$, *i.e.*, X receives at least an $\epsilon/2$ fraction of the service. □

Observe that this analysis holds for each good client separately. It follows that if the good clients deliver *in aggregate* an α fraction of the bandwidth, then *in aggregate* they will receive an $\alpha/2$ fraction of the service. Note that this claim remains true regardless of the service rate c, which need not be known to carry out the auction.

Theory versus practice. We now consider ways in which the above theorem is both weaker and stronger than what we expect to see in practice. We begin with weaknesses. First, consider the unreasonable assumption that requests are served with perfect regularity. The theorem can be trivially extended: for service times that fluctuate within a bounded range $[(1-\delta)/c, (1+\delta)/c]$ (as in our implementation; see §6), X receives at least a $(1-2\delta)\epsilon/2$ fraction of the service. However, even this looser restriction may be unrealistic in practice. And pathological service timings violate the theorem. For example, if many request fulfillments are bunched in a tiny interval during which X has not yet paid much, bad clients can cheaply outbid it during this interval, *if* they know that the pathology is happening and are able to time their bids. But doing so requires implausibly deep information.

Second, the theorem assumes that a good client "pays bytes" at a constant rate given by its bandwidth. However, the payment channel in our implementation runs over TCP, and TCP's slow start means that a good client's rate must grow. Moreover, because we implement the payment channel as a *series* of large HTTP POSTs (see §6), there is a quiescent period between POSTs (equal to two RTTs between client and thinner) as well as TCP's slow start for each POST. Nevertheless, we can extend the analysis to capture this behavior and again derive a lower bound for the fraction of service that a given good client receives. The result is that if the good client has a small fraction of the total bandwidth (causing it to spend a lot of time paying), and if the HTTP POST is big compared to the bandwidth-delay product, then the client's fraction of service is not noticeably affected (because the quiescent periods are negligible relative to the time spent paying at full rate).

We now consider the strength of the theorem: it makes no assumptions at all about adversarial behavior. We believe that in practice adversaries will attack the auction by opening many concurrent TCP connections to avoid quiescent periods, but the theorem handles every other case too. The adversary can open few or many TCP connections, disregard TCP semantics, or send continuously or in bursts. The only parameter in the theorem is the total number of bytes sent (in a given interval) by other clients.

The theorem does cede the adversary an extra factor of two "advantage" in bandwidth (the good client sees only $\epsilon/2$ service for ϵ bandwidth). This advantage arises because the proof lets the adversary control exactly when its bytes arrive—sending fewer when the good client's bid is small and more as the bid grows. This ability is powerful indeed—most likely stronger than real adversaries have. Nevertheless, even with this highly pessimistic assumption about adversarial abilities, speak-up can still do its job: the required provisioning has only increased by a factor of 2 over the ideal from §3.1—and this required provisioning is still far less than would be required to absorb the attack without speak-up.

In §7.4, we quantify the adversarial advantage in our experiments by determining how the factors mentioned in this section—quiescent periods for good clients, bad clients opening concurrent connections, etc.—affect the required provisioning above the ideal.

4 REVISITING ASSUMPTIONS

We have so far made a number of assumptions. Below we address four of them in turn: that aside from end-hosts' access links, the Internet has infinite capacity; that no bottleneck link is shared (which is a special case of the previous assumption, but we address it separately); that the thinner has infinite capacity; and that bad clients consume all of their upload bandwidth when they attack. In the next section, we relax the assumption of equal server requests.

4.1 Speak-up's Effect on the Network

No flow between a client and a thinner individually exhibits antisocial behavior. In our implementation, each payment channel comprises a series of HTTP POSTs (see §6) and thus inherits TCP's congestion control. (For UDP applications, the payment channel could use the congestion manager [3] or DCCP [22].) However, such individual courtesies do not automatically excuse the larger rudeness of increased traffic levels, and we must ask whether the network can handle this increase.

We give two sketchy arguments suggesting that speak-up would not much increase total traffic and then consider the effect of such increases. First, speak-up inflates *upload* bandwidth, and, despite the popularity of peer-to-peer file-sharing, most bytes still flow in the *download* direction [15]. Thus, inflating upload traffic even to the level of download traffic would cause an inflation factor of at most two. Second, only a very small fraction of Internet servers is attacked at any one time. Thus, even if speak-up did increase the traffic to each attacked site by an order of magnitude, the increase in overall Internet traffic would still be small.

Whatever the overall traffic increase, it is unlikely to be problematic for the Internet "core": both anecdotes from network operators and measurements [15] suggest that these links operate at low utilization. And, while the core cannot handle *every* client transmitting maximally (as argued in [46]), we expect that the fraction of clients doing so at any time will be small—again, because few sites will be attacked at any time. Speak-up will, however, create contention at bottleneck links, an effect that we explore experimentally in §7.7.

4.2 Shared Links

We now consider what happens when clients that share a bottleneck link are simultaneously encouraged by the thinner. For simplicity, assume two clients behind bottleneck link l; the discussion generalizes to more clients. If the clients are both good, their individual flows roughly share l, so they get roughly the same piece of the server. Each may be disadvantaged compared to clients that are not similarly bottlenecked, but neither is disadvantaged relative to the other. If, however, one of the clients is bad, then the good client has a problem: the bad client can open n parallel TCP connections (§3.4), claim roughly an $n/(n+1)$ fraction of l's bandwidth, and get a much larger piece of the server. While this outcome is unfortunate for the good client, observe, first, that the *server* is still protected (the bad client can "spend" at most l). Second, while the thinner's encouragement might instigate the bad client, the fact is that when a good and bad client share a bottleneck link—speak-up or no—the good client loses: the bad client can always deny service to the good client. We experimentally investigate such sharing in §7.6.

4.3 Provisioning the Thinner

For speak-up to work, the thinner must be uncongested: a congested thinner could not "get the word out" to encourage clients. Thus, the thinner needs enough bandwidth to absorb a full DDoS attack and more (which is condition C1 in §2.2). It also needs enough processing capacity to handle the dummy bytes. (Meeting this requirement is far easier than provisioning the *server* to handle the full attack because the thinner does not do much per-request processing.) We now argue that meeting these requirements is plausible.

One study of observed DoS attacks found that the 95th percentile of attack size is in the low hundreds of Mbits/s [38], which agrees with other anecdotes (*e.g.*, [45]). The traffic from speak-up would presumably be multiples larger since the good clients would also send at high rates. However, even with several Gbits/s of traffic in an attack, the thinner's requirements are not insurmountable.

First, providers readily offer links, even temporarily (*e.g.*, [33]), that accommodate these speeds. Such bandwidth is expensive, but co-located servers could share a thinner, or else the ISP could provide the thinner as a service (see condition C1 in §2.2). Second, we consider processing capacity. Our unoptimized software thinner running on commodity hardware can handle 1.5 Gbits/s of traffic and tens or even hundreds of thousands of concurrent clients; see §7.1. A production solution would presumably do much better.

4.4 Attackers' Constraints

The assumption that bad clients are today "maxing out" their *upload* bandwidth was made for ease of exposition. The required assumption is only that *bad clients consistently make requests at higher rates than legitimate clients*. Specifically, if bad clients are limited by their *download* bandwidth, or they are not maxed out at all today, speak-up is still useful: it *makes* upload bandwidth into a constraint by forcing everyone to spend this resource. Since bad clients—even those that aren't maxed out—are more active than good ones, the imposition of this upload bandwidth constraint affects the bad clients more, again changing the mix of the server that goes to the good clients. Our goals and analysis in §3 still hold: they are in terms of the bandwidth *available* to both populations, not the bandwidth that they actually *use* today.

5 HETEROGENEOUS REQUESTS

We now generalize the design to handle the more realistic case when the requests are unequal. We make the worst-case assumption that the thinner does not know their difficulty in advance but attackers do, as given by the threat model in §2.2. If the thinner treated all requests equally (charging, in effect, the average price for any request), an attacker could get a disproportionate share of the server by sending only the hardest requests.

In describing the generalization to the design, we make two assumptions:

- As in the homogeneous case, the server processes only one request at a time. Thus, the "hardness" of a computation is measured by how long it takes to complete. Relaxing this assumption to account for more complicated servers is not difficult, as long as the server implements processor sharing among concurrent requests, but we don't delve into those details here.
- The server exports an interface that allows the thinner to SUSPEND, RESUME, and ABORT requests. (Many transaction managers and application servers support such an interface.)

At a high level, the solution is for the thinner to break time into quanta, to see each request as comprising equal-sized *chunks* that consume a quantum of the server's attention, and to hold a virtual auction for each quantum. Thus, if a client's request is made of x chunks, the client must win x auctions for its request to be fully served. The thinner need not know x in advance for any request.

In more detail: rather than terminate the payment channel once the client's request is admitted (as in §3.3), the thinner extracts an *on-going* payment until the request completes. Given these on-going payments, the thinner implements the following procedure every τ seconds (τ is the quantum length):

1. Let v be the currently-active request. Let u be the contending request that has paid the most.
2. If u has paid more than v, then SUSPEND v, admit (or RESUME) u, and set u's payment to zero.
3. If v has paid more than u, then let v continue executing but set v's payment to zero (since v has not yet paid for the *next* quantum).
4. Time-out and ABORT any request that has been SUSPENDed for some period (*e.g.*, 30 seconds).

This scheme requires some cooperation from the server. First, the server should not SUSPEND requests that hold critical locks; doing so could cause deadlock. Second, SUSPEND, RESUME, and ABORT should have low overhead.

6 IMPLEMENTATION

We implemented a prototype thinner in C++ as an OKWS [23] Web service using the SFS toolkit [26]. It runs on Linux 2.6, exporting a well-known URL. When a Web client requests this URL, the thinner decides if, and when, to send this request to the server, using the method in §3.3. The server is currently emulated, running in the same address space as the thinner. The server "processes" requests with a "service time" selected uniformly at random from $[.9/c, 1.1/c]$. When the server responds to a request, the thinner returns HTML to the client with that response. Any JavaScript-capable Web browser can use our system; we have successfully tested our implementation with Firefox, Internet Explorer, Safari, and a custom client that we use in our experiments.

Whenever the emulated server is not free, the thinner returns JavaScript to the Web client that causes it to automatically issue *two* HTTP requests: (1) the actual request to the server, and (2) a one-megabyte HTTP POST that is dynamically constructed by the browser and that holds dummy data (one megabyte reflecting some browsers' limits on POSTs). The thinner delays responding to the first HTTP request (because the response to that request has to come from the server, which is busy). The second HTTP request is the payment channel. If, while sending these dummy bytes, the client wins the auction, the thinner terminates request (2) and gives request (1) to the server. If, on the other hand, request (2) completes, the client has not yet received service; in this case, the thinner returns JavaScript that causes the browser to send another large POST, and the process continues. The thinner correlates the client's payments with its request via an "id" field in both HTTP requests.

One can configure the thinner to support hundreds of thousands of concurrent connections by setting the maximum number of connection descriptors appropriately. (The thinner evicts old clients as these descriptors deplete.) With modern versions of Linux, the limit on concurrent clients is not per-connection descriptors but rather the RAM consumed by each open connection.

7 EXPERIMENTAL EVALUATION

To investigate the effectiveness and performance of speak-up, we conducted experiments with our prototype thinner. Our primary question is how the thinner allocates an attacked server to good clients. To answer this question, we begin in §7.2 by varying the bandwidth of good (G) and bad (B) clients, and measuring how the server is allocated with and without speak-up. We also measure this allocation with server capacities above and below the ideal in §3.1. In §7.3, we measure speak-up's latency and byte cost. In §7.4, we ask how much bad clients can "cheat" speak-up to get more than a bandwidth-proportional share of the server. §7.5 shows how speak-up performs when clients have differing bandwidths and latencies to the thinner. We also explore scenarios in which speak-up traffic shares a bottleneck link with other speak-up traffic (§7.6) and with non-speak-up traffic (§7.7). Table 1 summarizes our results.

Our thinner implementation allocates the emulated server in rough proportion to clients' bandwidths.	§7.2, §7.5
In our experiments, the server needs to provision only 15% beyond the bandwidth-proportional ideal to serve all good requests.	§7.3, §7.4
Our unoptimized thinner implementation can sink 1.5 Gbits/s of uploaded "payment traffic".	§7.1
On a bottleneck link, speak-up traffic can crowd out other speak-up traffic and non-speak-up traffic.	§7.6, §7.7

Table 1: Summary of main evaluation results.

7.1 Setup and Method

All of the experiments described here ran on the Emulab testbed [13]. The clients run a custom Python Web client and connect to the prototype thinner in various emulated topologies. The thinner runs on Emulab's "PC 3000", which has a 3 GHz Xeon processor and 2 GBytes of RAM; the clients are allowed to run on any of Emulab's hardware classes.

All experiments run for 600 seconds. Each client runs on a separate Emulab host and generates *requests*. A request proceeds as follows. The client first makes the actual request to the server. If the server is busy, the thinner replies and makes the client issue two HTTP requests: the original request and the payment bytes. Each client's requests are driven by a Poisson process of rate λ requests/s. However, a client never allows more than a configurable number w (the window) of outstanding requests. If the stochastic process "fires" when more than w requests are outstanding, the client puts the new request in a backlog queue, which drains when the client receives a response to an earlier request. If a request is in this queue for more than 10 seconds, it times out, and the client logs a service denial. All requests are identical, and the server itself is emulated, processing a request on average every $1/c$ seconds (see §6).

We use the behavior just described to model both good and bad clients. A bad client, by definition, tries to capture more than its fair share. We model this intent as follows: in our experiments, bad clients send requests faster than good clients, and bad clients send requests concurrently. Specifically, we choose $\lambda = 40$, $w = 20$ for bad clients and $\lambda = 2$, $w = 1$ for good clients. (The w value for bad clients is pessimistic; see §7.4.)

Our choices of B and G are determined by the number of clients that we are able to run in the testbed and by a rough model of today's client access links. Specifically, in most of our experiments, there are 50 clients, each with 2 Mbits/s of access bandwidth. Thus, $B + G$ usually equals 100 Mbits/s. This scale is smaller than most attacks. Nevertheless, we believe that the results generalize because we focus on how the prototype's behavior differs from the theory in §3. By understanding this difference, one can can make predictions about speak-up's performance in larger attacks.

Because the experimental scale does not tax the thinner, we separately measured its capacity and found that it can handle loads comparable to recent attacks. At 90% CPU utilization on the hardware described above with multiple gigabit Ethernet interfaces, in a 600-second experiment with a time series of 5-second intervals, the thinner sinks payment bytes at 1451 Mbits/s (with standard deviation of 38 Mbits/s) for 1500-byte packets and at 379 Mbits/s (with standard deviation of 24 Mbits/s) for 120-byte packets. Many re-

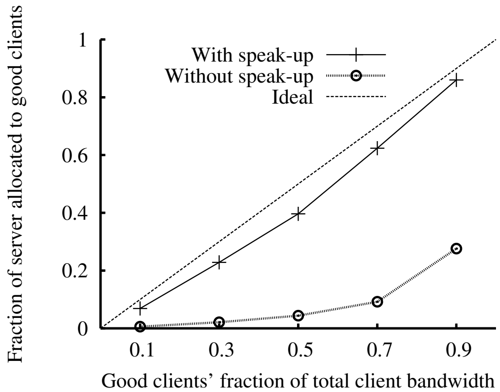

Figure 2: Server allocation when $c = 100$ requests/s as a function of $\frac{G}{G+B}$. The measured results for speak-up are close to the ideal line. Without speak-up, bad clients sending at $\lambda = 40$ requests/s and $w = 20$ capture much more of the server.

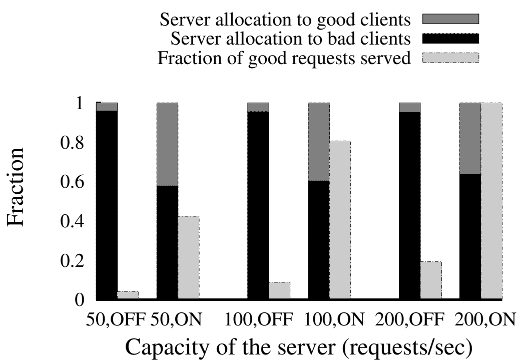

Figure 3: Server allocation to good and bad clients, and the fraction of good requests that are served, without ("OFF") and with ("ON") speak-up. c varies, and $G = B = 50$ Mbits/s. For $c = 50, 100$, the allocation is roughly proportional to the aggregate bandwidths, and for $c = 200$, all good requests are served.

cent attacks are roughly this size; see §2.1 and §4.3. The capacity also depends on how many concurrent clients the thinner supports; the limit here is only the RAM for each connection (see §6).

7.2 Validating the Thinner's Allocation

When the rate of incoming requests exceeds the server's capacity, speak-up's goal is to allocate the server's resources to a group of clients in proportion to their aggregate bandwidth. In this section, we evaluate to what degree our implementation meets this goal.

In our first experiment, 50 clients connect to the thinner over a 100 Mbits/s LAN. Each client has 2 Mbits/s of bandwidth. We vary f, the fraction of "good" clients (the rest are "bad"). In this homogeneous setting, $\frac{G}{G+B}$ (*i.e.*, the fraction of "good client bandwidth") equals f, and the server's capacity is $c = 100$ requests/s.

Figure 2 shows the fraction of the server allocated to the good clients as a function of f. Without speak-up, the bad clients capture a larger fraction of the server than the good clients because they make more requests and the server, when overloaded, randomly drops requests. With speak-up, however, the good clients can "pay" more for each of their requests—because they make fewer—and can thus capture a fraction of the server roughly in proportion to their bandwidth. The small difference between the measured and ideal values is a result of the good clients not using as much of their bandwidth as the bad clients. We discussed this adversarial advantage in §3.4 and further quantify it in §7.3 and §7.4.

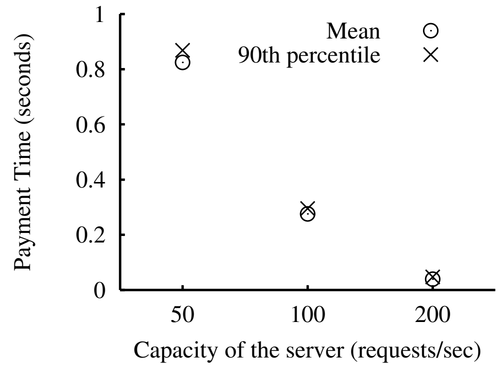

Figure 4: Mean time to upload dummy bytes for good requests that receive service. c varies, and $G = B = 50$ Mbits/s. When the server is not overloaded ($c = 200$), speak-up introduces little latency.

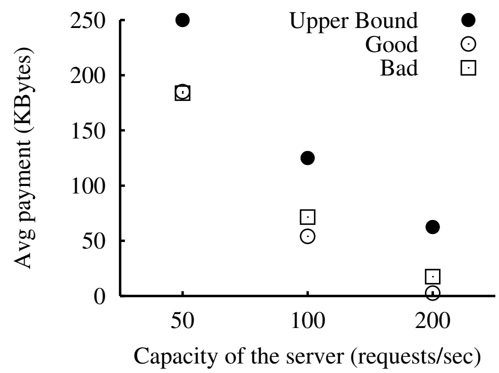

Figure 5: Average number of bytes sent on the payment channel—the "price"—for served requests. c varies, and $G = B = 50$ Mbits/s. When the server is overloaded ($c = 50, 100$), the price is close to the upper bound, $(G + B)/c$; see the text for why they are not equal. When the server is not overloaded ($c = 200$), good clients pay almost nothing.

In the next experiment, we investigate different "provisioning regimes". We fix G and B, and measure the server's allocation when its capacity, c, is less than, equal to, and greater than c_{id}. Recall from §3.1 that c_{id} is the minimum value of c at which all good clients get service, if speak-up is deployed and if speak-up allocates the server exactly in proportion to client bandwidth. We set $G = B$ by configuring 50 clients, 25 good and 25 bad, each with a bandwidth of 2 Mbits/s to the thinner over a LAN. In this scenario, $c_{id} = 100$ requests/s (from §3.1, $c_{id} = g(1 + \frac{B}{G}) = 2g = 2 \cdot 25 \cdot \lambda = 100$), and we experiment with $c = 50, 100, 200$ requests/s.

Figure 3 shows the results. The good clients get a larger fraction of the server with speak-up than without. Moreover, for $c = 50, 100$, the allocation under speak-up is roughly proportional to the aggregate bandwidths, and for $c = 200$, all good requests are served. Again, one can see that the allocation under speak-up does not exactly match the ideal: from Figure 3, when speak-up is enabled and $c = c_{id} = 100$, the good demand is not fully satisfied.

7.3 Latency and Byte Cost

We now explore the byte and latency cost of speak-up for the same set of experiments (c varies, 50 clients, $G = B = 50$ Mbits/s). For the latency cost, we measure the length of time that clients spend uploading dummy bytes, which captures the extra latency that speak-up introduces. Figure 4 shows the averages and 90th percentiles of these measurements for the served good requests.

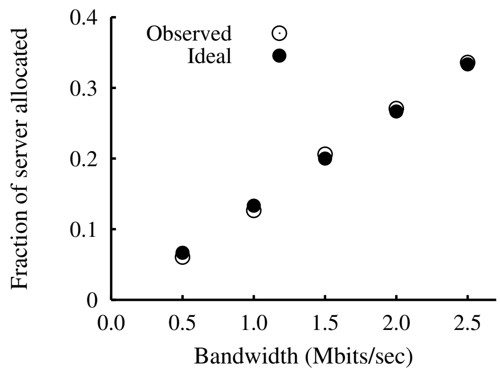

Figure 6: Heterogeneous client bandwidth experiments with 50 LAN clients, all good. The fraction of the server ($c = 10$ requests/s) allocated to the ten clients in category i, with bandwidth $0.5 \cdot i$ Mbits/s, is close to the ideal proportional allocation.

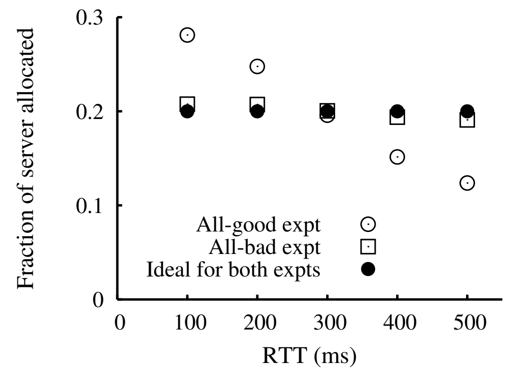

Figure 7: Two sets of heterogeneous client RTT experiments with 50 LAN clients, all good or all bad. The fraction of the server ($c = 10$ requests/s) captured by the 10 clients in category i, with RTT $100 \cdot i$ ms, varies for good clients. In contrast, bad clients' RTTs don't matter because they open multiple connections.

For the byte cost, we measure the number of bytes uploaded for served requests—the "price"—as recorded by the thinner. Figure 5 shows the average of this measurement for good and bad clients and also plots the theoretical average price, $(G + B)/c$, from §3.3, which is labeled "Upper Bound". The actual price is lower than this theoretical one because the clients do not consume all of their bandwidth, for reasons that we now describe. We consider the different values of c in turn.

For $c = 50$, each good client spends an average of 1.25 Mbits/s (determined by tallying the total bits spent by good clients over the experiment). This average is less than the 2 Mbits/s access link because of a quiescent period between when a good client first issues a request and when the thinner replies, asking for payment. This period is roughly 0.35 seconds, the length owing to a long backlog at the thinner of requests and payment bytes. When not in a quiescent period, a good client consumes most of its access link, delivering 1.8 Mbits/s on average, inferred by dividing the average good client payment (Figure 5) by the average time spent paying (Figure 4).

Bad clients, in contrast, keep multiple requests outstanding so do not have "down time". For $c = 50$, their payments are 1.7 Mbits/s on average. They actually deliver slightly more than this number but occasionally "waste" bytes. This wastage happens when a bad client establishes a payment channel but—because its outbound bandwidth is nearly fully utilized—fails to deliver the accompanying request. Meanwhile, the thinner accepts payment for 10 seconds, at which point it times out the payment channel.

The $c = 100$ case is similar to $c = 50$, except bad clients see a higher price than good ones. The reason is as follows. Bad clients waste bytes, as just described. In this case, however, some of the requests actually arrive before the 10 seconds have elapsed—but long after the client has paid enough to win the auction. In those instances, bad clients overpay hugely, increasing their average price.

For $c = 200$, clients do not have to pay much because the server is lightly loaded. In fact, good and bad clients often encounter a price of zero, though bad clients again overpay sometimes.

7.4 Empirical Adversarial Advantage

As just discussed, bad clients are able to deliver more bytes than good clients in our experiments. As a result of this disparity, the server does not achieve the ideal of a bandwidth-proportional allocation. This effect was visible in §7.2.

To better understand this adversarial advantage, we ask, What is the minimum value of c at which all of the good demand is satisfied? To answer this question, we experimented with the same configuration as above ($G = B = 50$ Mbits/s; 50 clients) but for more values of c. We found that all of the good demand is satisfied at $c = 115$, which is only 15% more provisioning than c_{id}, the capacity needed under exact proportional allocation. We conclude that a bad client can cheat the proportional allocation mechanism but only to a limited extent—at least under our model of bad behavior.

We now revisit that model. We chose $w = 20$ to be conservative: for other values of w between 1 and 60 (again, $B = G$, $c = 100$), the bad clients capture less of the server. (We hypothesize that for $w > 20$, the damage from wasted bytes exceeds the benefit from no quiescence.) However, the qualitative model does have weaknesses. For example, our bad clients sometimes overpay (as discussed in §7.3), and a truly pessimal bad client would not. Nevertheless, the analysis in §3.4 shows that bad clients cannot do much better than the naïve behavior that we model.

7.5 Heterogeneous Network Conditions

We now investigate the server's allocation for different client bandwidths and RTTs. We begin with bandwidth. We assign 50 clients to 5 categories. The 10 clients in category i ($1 \leq i \leq 5$) have bandwidth $0.5 \cdot i$ Mbits/s and are connected to the thinner over a LAN. All clients are good. The server has capacity $c = 10$ requests/s. Figure 6 shows that the resulting server allocation to each category is close to the bandwidth-proportional ideal.

We now consider RTT, hypothesizing that the RTT between a good client and the thinner will affect the allocation, for two reasons. First, at low prices, TCP's ramp-up means that clients with longer RTTs will take longer to pay. Second, and more importantly, each request has at least one associated quiescent period (see §7.1 and §7.3), the length of which depends on RTT. In contrast, bad clients have multiple requests outstanding so do not have "down time" and will not be much affected by their RTT to the thinner.

To test this hypothesis, we assign 50 clients to 5 categories. The 10 clients in category i ($1 \leq i \leq 5$) have RTT $= 100 \cdot i$ ms to the thinner, giving a wide range of RTTs. All clients have bandwidth 2 Mbits/s, and $c = 10$ requests/s. We experiment with two cases: all clients good and all bad. Figure 7 confirms our hypothesis: good clients with longer RTTs get a smaller share of the server while for bad clients, RTT matters little. This result may seem unfortunate, but the effect is limited: for example, in this experiment, no good client gets more than double or less than half the ideal.

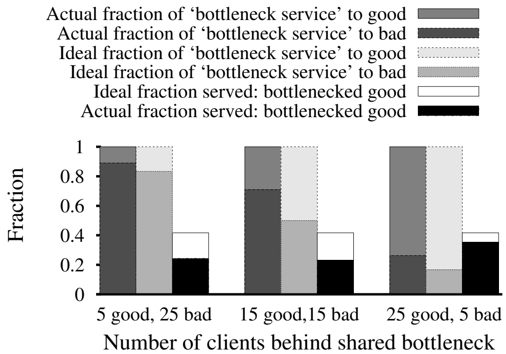

Figure 8: Server allocation when good and bad clients share a bottleneck link, *l*. "Bottleneck service" refers to the portion of the server captured by all of the clients behind *l*. The actual breakdown of this portion (left bar) is worse for the good clients than the bandwidth-proportional allocation (middle bar) because bad clients "hog" *l*. The right bar further quantifies this effect.

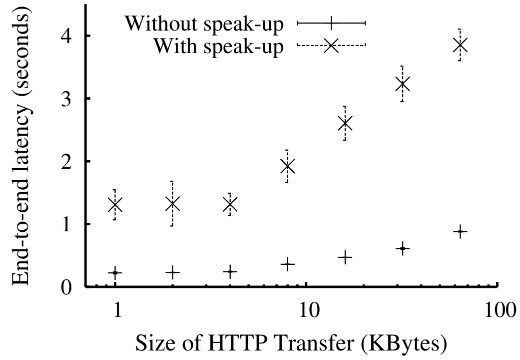

Figure 9: Effect on an HTTP client of sharing a bottleneck link with speak-up clients. Graph shows means and standard deviations of end-to-end HTTP download latency with and without speak-up running, for various HTTP transfer sizes (which are shown on a log scale).

7.6 Good and Bad Clients Sharing a Bottleneck

When good clients share a bottleneck link with bad ones, good requests can be "crowded out" by bad ones before reaching the thinner (see §4.2). We quantify this observation with an experiment that uses the following topology: 30 clients, each with a bandwidth of 2 Mbits/s, connect to the thinner through a common link, *l*. The bandwidth of *l* is 40 Mbits/s. *l* is a bottleneck because the clients behind *l* can generate 60 Mbits/s. Also, 10 good and 10 bad clients, each with a bandwidth of 2 Mbits/s, connect to the thinner directly through a LAN. The server's capacity is $c = 50$ requests/s. We vary the number of good and bad clients behind *l*.

In all cases, the clients behind *l* together capture half of the server's capacity (as expected, given the topology). We measure how this "server half" is allocated to the good and bad clients behind *l*. We also measure, of the good requests that originate behind *l*, what fraction receive service. Figure 8 depicts these measurements and compares them to the bandwidth-proportional ideals.[2] The effect on good clients, visible in the figure, will likely be more pronounced when the bottleneck's bandwidth is a smaller fraction of the combined bandwidth behind it.

7.7 Impact of Speak-up on Other Traffic

We now consider how speak-up affects other traffic, specifically what happens when a TCP endpoint, *H*, shares a bottleneck link, *m*, with clients that are currently uploading dummy bytes. The case when *H* is a TCP sender is straightforward: *m* will be shared among *H*'s transfer and the speak-up uploads. When *H* is a TCP receiver, the extra traffic from speak-up affects *H* in two ways. First, ACKs from *H* will be lost (and delayed) more often than without speak-up. Second, for request-response protocols (*e.g.*, HTTP), *H*'s request can be delayed. Here, we investigate these effects on HTTP downloads.

We experiment with the following setup: 10 good speak-up clients share a bottleneck link, *m*, with *H*, a host that runs the HTTP client `wget`. *m* has a bandwidth of 1 Mbit/s and one-way delay 100 ms. Each of the 11 clients has a bandwidth of 2 Mbits/s. On the other side of *m* are the thinner (fronting a server with $c = 2$ requests/s) and a separate Web server, S. In each experiment, *H* downloads a file from S 100 times.

Figure 9 shows the means and standard deviations of the download latency for various file sizes, with and without the speak-up traffic. There is significant "collateral damage" to "innocently bystanding" Web transfers here: download times inflate by almost 6× for a 1 Kbyte (single packet) transfer and by almost 4.5× for 64 Kbyte transfers. However, this experiment is quite pessimistic: the RTTs are large, the bottleneck bandwidth is highly restrictive (roughly 20× smaller than the demand), and the server capacity is low. While speak-up is clearly the exacerbating factor in this experiment, speak-up will not have this effect on every link.

[2] For the first measurement, the ideal is simply the fraction of good and bad clients behind *l*. For the second measurement, the ideal presumes that the non-bottlenecked clients each have 2 Mbits/s of bandwidth and that the clients behind *l* have $2(\frac{40}{60})$ Mbits/s.

8 RELATED WORK

In this section, we first survey related work in the context of comparing speak-up to other defenses against *application-level* DDoS attacks. (For other attacks and defenses, see the survey by Mirkovic and Reiher [28] and the bibliographies in [21,29,51].) We then discuss how and when to combine speak-up with other defenses.

8.1 Comparisons to Related Work

Using the taxonomy in §1 (massive over-provisioning, detect and block, currency), speak-up is a currency scheme. The currency concept was pioneered by Dwork and Naor [12] in the context of spam defense. Others have done work in the same spirit [1,6,7,11,20,25, 49]; these approaches are often called proof-of-work schemes.

We first proposed bandwidth as a currency in a workshop paper [48]. In contrast to [48], this paper gives a viable mechanism and an implementation, evaluation, and analysis of that mechanism; presents a solution to the "unequal requests" case; and considers context and alternate DDoS defenses much more completely.

We do not know of another proposal to use bandwidth as a currency. However, the authors of [17, 39] describe a solution to DoS attacks on servers' computational resources in which good clients send a fixed number of copies of their messages and the server only processes a fixed fraction of the messages that it receives, thereby diminishing adversaries' impact. Our work shares an ethos but has a very different realization. In that work, the drop probability and repeat count are hard-coded, and the approach does not apply to HTTP. Further, the authors do not consider congestion control, the implications of deployment in today's Internet, and the unequal requests case. Also, Gligor [16] observes that client retries and timeouts require less overhead while still providing the same qualitative performance bounds as proof-of-work schemes. Because the general approach does not meet his more exacting performance requirements, he does not consider using bandwidth as currency.

Although we do not claim that bandwidth is strictly better than other currencies, we do think it is particularly natural. With other currencies, the server must either report an explicit price (*e.g.*, by sending a puzzle with a specific hardness) or have the clients guess the price. With speak-up, in contrast, this function happens automatically: the correct price emerges, and neither the thinner nor the client has to know the price in advance.

The drawbacks of currency-based schemes are, first, that the good clients must have enough currency [24] (*e.g.*, speak-up only applies when the good clients have enough bandwidth) and, second, that the currency can be unequally distributed (*e.g.*, some clients have faster uplinks than others). We discuss this latter disadvantage in §9. Another critique of currency schemes is that they give attackers *some* service so might be weaker than the schemes we discuss below (such as profiling) that seek to *block* attackers. However, under those schemes, a smart bot can imitate a good client, succeed in fooling the detection discipline, and again get *some* service.

The most commonly deployed defense [30] is a combination of link over-provisioning [33] and profiling, which is a detect-and-block approach offered by several vendors [5, 9, 27]. These latter products build a historical profile of the defended server's clientele and, when the server is attacked, block traffic violating the profile. Many other detect-and-block schemes have been proposed; we now mention a few. Resource containers [8] perform rate-limiting to allocate the server's resources to clients fairly. Defenses based on CAPTCHAs [47] (*e.g.*, [29, 42]) use reverse Turing tests to block bots. Killbots [21] combines CAPTCHAs and rate-limiting, defining a bot as a non-CAPTCHA answering host that sends too many requests to an overloaded server. With capabilities [4, 50, 51], the network blocks traffic not authorized by the application; to decide which traffic to authorize, the application can use rate-limiting, CAPTCHAs, or other rules.

One critique of detect-and-block methods is that they can err. CAPTCHAs can be thwarted by "bad humans" (cheap labor hired to attack a site or induced [32] to solve the CAPTCHAs) or "good bots" (legitimate, non-human clientele or humans who do not answer CAPTCHAs). Schemes that rate-limit clients by IP address can err with NAT (a large block of customers is rate-limited as one customer) or spoofing (a small number of clients can get a large piece of the server). Profiling apparently addresses some of these shortcomings today (*e.g.*, many legitimate clients behind a NAT would cause the NAT's external IP address to have a higher baseline rate in the server's profile). However, in principle such "behavior-based" techniques can also be "fooled": a set of savvy bots could, over time, "build up" their profile by appearing to be legitimate clients, at which point they could abuse their profile and attack.

8.2 Combining with Related Work

Practical DDoS defense involves composing various methods from the taxonomy in §1. We do not outline a complete DDoS protection strategy here but only discuss how to protect two classes of resources. First, all sites, whether using speak-up or not, must defend their access links from saturation. Speak-up in particular requires that the thinner is not congested (§4.3). The best current strategy for link defense seems to be a combination of over-provisioning (*e.g.*, [33]), blocking obviously spurious traffic (*e.g.*, ICMP floods), and shaping "in-band" traffic via historical profiling (*e.g.*, [5,9,27]).

Second, sites with scarce computational resources must implement application-level defense. Given that profiling is required to protect the link anyway, we must ask when it suffices as an application-level defense. Our answer is when the following conditions all hold: no pre-defined clientele (C3 from §2.2); non-human clientele (C4); and the negation of C5, *i.e.*, when requests cause equal amounts of work, when spoofing is implausible, and when bots trigger alarms. We now briefly consider what to do when the conditions for profiling are not met. When C3 doesn't hold, one can use capabilities [4,50,51] or explicit filters. When C4 doesn't hold, one may be able to use CAPTCHAs to preferentially admit humans. And of course, when C5 *does* hold, and when C1 and C2 do too, we advocate speak-up as the application-level DDoS defense.

9 OBJECTIONS

Even under the conditions when speak-up is most applicable, it may still raise objections, some of which we now address.

Bandwidth envy. Before speak-up, all good clients competed equally for a small share of the server. Under speak-up, more good clients are "better off" (*i.e.*, can claim a larger portion of the server). But since speak-up allocates the server's resources in proportion to a client's bandwidth, high-bandwidth good clients are "more better off", and this inequality might be problematic. However, observe that unfairness only occurs under attack. Thus, while we think this inequality is unfortunate, it is not fatal. A possible solution is for ISPs with low-bandwidth customers to offer access to high-bandwidth proxies whose purpose is to "pay bandwidth" to the thinner. These proxies would have to allocate *their* resources fairly—perhaps by implementing speak-up recursively.

Variable bandwidth costs. In some countries, customers pay their ISPs "per-bit". For those customers, access to a server defended by speak-up (and under attack) would cost more than usual. One possible solution is the proxy mentioned above. Another one is a "price tag": the thinner would expose the "going rate" in bytes, and the ISP would translate this figure to money and report it to customers, letting them choose whether to pay for access.

Incentives for ISPs. One might ask whether speak-up gives ISPs an incentive to encourage botnets as a way to increase the bandwidth demanded by good clients. Our response is that such misalignment of incentives can happen in many commercial relationships (*e.g.*, investment managers who needlessly generate commissions), but society relies on a combination of regulation, professional norms, and reputation to limit harmful conduct.

Solving the wrong problem. One might ask, "If the problem is bots, then shouldn't researchers address that mess instead of encouraging more traffic?" Our answer to this philosophical question is that cleaning up bots is crucial, but even if bots are curtailed by orders of magnitude, a server with scarce computational resources must still limit bots' influence. Speak-up is a way to do so.

Flash crowds. Speak-up treats a flash crowd (overload from good clients alone) just like an application-level DDoS attack. This fact might appear unsettling. Observe, however, that it does not apply to the canonical case of a flash crowd, in which a hyperlink from `slashdot.org` overwhelms a residential Web site's access link: speak-up would not have been deployed to defend a low-bandwidth site (see §2.2). For sites in our applicability regime, making good clients "bid" for access when *all* clients are good is certainly not ideal, but the issues here are the same as with speak-up in general.

10 CONCLUSION

This study has sought to answer two high-level questions: (1) Which conditions call for speak-up's peculiar brand of protection? (2) Does speak-up admit a practical design? Notably absent from this list is a question about how often the conditions in (1) do and will hold, *i.e.*, who needs speak-up? To answer that question definitively will require not just a measurement effort but also a broader "market survey"—a survey about demand that, to be credible, will have to gather the opinions of network operators, server operators, and even users. Rather than trying to see who would buy—which we plan to do next—we decided first to see what we could build. Perhaps our priorities were inverted. Nevertheless, we report our main finding: based on the design, analysis, and evaluation of a prototype and subject to much future work and many issues, we can give a cautiously affirmative answer to question (2).

Acknowledgments

We thank the HotNets 2005 attendees, especially Nick Feamster, Vern Paxson, Adrian Perrig, and Srini Seshan, for important critiques of our approach; Frans Kaashoek, Max Krohn, Sara Su, Arvind Thiagarajan, Keith Winstein, and the anonymous reviewers, both regular and shadow PC, for excellent comments on drafts; Ben Adida, Dave Andersen, Micah Brodsky, Russ Cox, Jon Crowcroft, Nick Feamster, Sachin Katti, Eddie Kohler, Christian Kreibich, Max Poletto and Andrew Warfield, for useful conversations; and Emulab [13]. This work was supported by the NSF under grants CNS-0225660 and CNS-0520241, by an NDSEG Graduate Fellowship, and by British Telecom.

References

[1] M. Abadi, M. Burrows, M. Manasse, and T. Wobber. Moderately hard, memory-bound functions. In *NDSS*, 2003.
[2] S. Agarwal, T. Dawson, and C. Tryfonas. DDoS mitigation via regional cleaning centers. Sprint ATL Research Report RR04-ATL-013177, Aug. 2003.
[3] D. G. Andersen et al. System support for bandwidth management and content adaptation in Internet applications. In *OSDI*, Sept. 2000.
[4] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet denial-of-service with capabilities. In *HotNets*, Nov. 2003.
[5] Arbor Networks, Inc. http://www.arbornetworks.com.
[6] T. Aura, P. Nikander, and J. Leiwo. DoS-resistant authentication with client puzzles. In *Intl. Wkshp. on Security Prots.*, 2000.
[7] A. Back. Hashcash. http://www.cypherspace.org/adam/hashcash/.
[8] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, Feb. 1999.
[9] Cisco Guard, Cisco Systems, Inc. http://www.cisco.com.
[10] Criminal Complaint: USA v. Ashley, Hall, Schictel, Roby, and Walker, Aug. 2004. http://www.reverse.net/operationcyberslam.pdf.
[11] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam. In *CRYPTO*, 2003.
[12] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.
[13] Emulab. http://www.emulab.net.
[14] N. Feamster, J. Jung, and H. Balakrishnan. An empirical study of "bogon" route advertisements. *CCR*, 35(1), Jan. 2005.
[15] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and C. Diot. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Network*, 17(6), 2003.
[16] V. D. Gligor. Guaranteeing access in spite of distributed service-flooding attacks. In *Intl. Wkshp. on Security Prots.*, 2003.
[17] C. A. Gunter, S. Khanna, K. Tan, and S. Venkatesth. DoS protection for reliably authenticated broadcast. In *NDSS*, 2004.
[18] M. Handley. Internet architecture WG: DoS-resistant Internet subgroup report, 2005. http://www.communicationsresearch.net/dos-resistant/meeting-1/cii-dos-summary.pdf.
[19] Honeynet Project and Research Alliance. Know your enemy: Tracking botnets. Mar. 2005. http://www.honeynet.org/papers/bots/.
[20] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *NDSS*, 1999.
[21] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds. In *USENIX NSDI*, May 2005.
[22] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: Congestion control without reliability. In *SIGCOMM*, Sept. 2006.
[23] M. Krohn. Building secure high-performance Web services with OKWS. In *USENIX Technical Conference*, June 2004.
[24] B. Laurie and R. Clayton. "Proof-of-Work" proves not to work; version 0.2, Sept. 2004. http://www.cl.cam.ac.uk/users/rnc1/proofwork2.pdf.
[25] D. Mankins, R. Krishnan, C. Boyd, J. Zao, and M. Frentz. Mitigating distributed denial of service attacks with dynamic resource pricing. In *Proc. IEEE ACSAC*, Dec. 2001.
[26] D. Mazières. A toolkit for user-level file systems. In *USENIX Technical Conference*, June 2001.
[27] Mazu Networks, Inc. http://mazunetworks.com.
[28] J. Mirkovic and P. Reiher. A taxonomy of DDoS attacks and DDoS defense mechanisms. *CCR*, 34(2), Apr. 2004.
[29] W. Morein, A. Stavrou, D. Cook, A. Keromytis, V. Mishra, and D. Rubenstein. Using graphic turing tests to counter automated DDoS attacks against Web servers. In *ACM CCS*, Oct. 2003.
[30] *Network World*. Extortion via DDoS on the rise. May 2005. http://www.networkworld.com/news/2005/051605-ddos-extortion.html.
[31] K. Park, V. S. Pai, K.-W. Lee, and S. Calo. Securing Web service by automatic robot detection. In *USENIX Technical Conference*, June 2006.
[32] *Pittsburgh Post-Gazette*. CMU student taps brain's game skills. Oct. 5, 2003. http://www.post-gazette.com/pg/03278/228349.stm.
[33] Prolexic Technologies, Inc. http://www.prolexic.com.
[34] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *SIGCOMM*, Sept. 2006.
[35] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the Internet. In *SIGCOMM*, Aug. 2004.
[36] E. Ratliff. The zombie hunters. The New Yorker, Oct. 10, 2005.
[37] SecurityFocus. FBI busts alleged DDoS mafia. Aug. 2004. http://www.securityfocus.com/news/9411.
[38] V. Sekar, N. Duffield, O. Spatscheck, J. van der Merwe, and H. Zhang. LADS: Large-scale automated DDoS detection system. In *USENIX Technical Conference*, June 2006.
[39] M. Sherr, M. Greenwald, C. A. Gunter, S. Khanna, and S. S. Venkatesh. Mitigating DoS attack through selective bin verification. In *1st Wkshp. on Secure Netwk. Protcls.*, Nov. 2005.
[40] K. K. Singh. Botnets—An introduction, 2006. http://www-static.cc.gatech.edu/classes/AY2006/cs6262_spring/botnets.ppt.
[41] Spammer-X. *Inside the SPAM Cartel*. Syngress, 2004. Page 40.
[42] Stupid Google virus/spyware CAPTCHA page. http://www.spy.org.uk/spyblog/2005/06/stupid_google_virusspyware_cap.html.
[43] TechWeb News. Dutch botnet bigger than expected. Oct. 2005. http://informationweek.com/story/showArticle.jhtml?articleID=172303265.
[44] *The Register*. East European gangs in online protection racket. Nov. 2003.
[45] D. Thomas. Deterrence must be the key to avoiding DDoS attacks, 2005. http://www.vnunet.com/computing/analysis/2137395/deterrence-key-avoiding-ddos-attacks.
[46] R. Vasudevan, Z. M. Mao, O. Spatscheck, and J. van der Merwe. Reval: A tool for real-time evaluation of DDoS mitigation strategies. In *USENIX Technical Conference*, June 2006.
[47] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. *CACM*, 47(2), Feb. 2004.
[48] M. Walfish, H. Balakrishnan, D. Karger, and S. Shenker. DoS: Fighting fire with fire. In *HotNets*, Nov. 2005.
[49] X. Wang and M. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *IEEE Symp. on Security and Privacy*, May 2003.
[50] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks. In *IEEE Symp. on Security and Privacy*, May 2004.
[51] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *SIGCOMM*, Aug. 2005.