DDoS Defense by Offense

MICHAEL WALFISH
UT Austin
MYTHILI VUTUKURU, HARI BALAKRISHNAN, and DAVID KARGER
MIT CSAIL
and
SCOTT SHENKER
UC Berkeley and ICSI

3

This article 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 so can 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 bandwidths, which is the intended result.

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

General Terms: Design, Experimentation, Security

Additional Key Words and Phrases: DoS attack, bandwidth, currency

ACM Reference Format:

Walfish, M., Vutukuru, M., Balakrishnan, H., Karger, D., and Shenker, S. 2010. DDoS defense by offense. ACM Trans. Comput. Syst. 28, 1, Article 3 (March 2010), 54 pages.
DOI = 10.1145/1731060.1731063 http://doi.acm.org/10.1145/1731060.1731063

This work was supported by the ONR under grant N00014-09-10757, by the NSF under grants CNS-0225660 and CNS-0520241, by an NDSEG Graduate Felloship, by an NSF Graduate Fellowship, and by British Telecom.
Part of this work was done when M. Walfish was at MIT.
Corresponding author's address: M. Walfish, Department of Computer Science, The University of Texas at Austin, 1 University Station C0500, Taylor Hall 2.124, Austin, TX 78712-0233; email: mwalfish@ cs.utexas.edu.

DOI 10.1145/1731060.1731063 http://doi.acm.org/10.1145/1731060.1731063

1. INTRODUCTION

This article is about a defense to *application-level* Distributed Denial of Service (DDoS), a particularly noxious attack in which computer criminals mimic legitimate client behavior by sending proper-looking requests, often via compromised and commandeered hosts [Ratliff 2005; SecurityFocus 2004; cyberslam 2004; Handley 2005], known as *bots*. By exploiting the fact that many Internet servers have "open clientele"—that is, they cannot tell whether a client is good or bad 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 link flood or TCP SYN flood is twofold. First, far less bandwidth is required: 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 [Register 2003; Network World 2005]) attacks include HTTP requests for large files [SecurityFocus 2004; Ratliff 2005], making queries of search engines [cyberslam 2004], and issuing computationally expensive requests (e.g., database queries or transactions) [Kandula et al. 2005].

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 tactics are different. We ask *all clients to speak up*, rather than sit idly by while attackers drown them out. We rely on *encouragement* (a term made precise in Section 4.1), whereby the server causes a client, resources permitting, to automatically send a higher volume of traffic. The key insight of this defense is as follows: we suppose that bad clients are already using most of their upload bandwidth, so they cannot react to the encouragement, whereas good clients have spare upload bandwidth, so they can send drastically higher volumes of traffic. As a result, the traffic into the server inflates, but the good clients are much better represented in the traffic mix than before and thus capture a much larger fraction of the server's resources than before.

1.1 Justification and Philosophy

To justify this approach at a high level and to explain our philosophy, we now discuss three categories of defense. The first approach that one might consider is to *overprovision massively*: in theory, one could purchase enough computational resources to serve both good clients and attackers. However, anecdotal evidence [Google Captcha 2005; Network World 2005] suggests that while some sites provision additional *link* capacity during attacks using commercial services[1,2], even the largest Web sites try to conserve *computation* by detecting bots and denying them access, using the methods in the second category.

[1]http://www.prolexic.com.
[2]http://www.counterpane.com/ddos-offerings.html.

We call this category—approaches that try to distinguish between good and bad clients—*detect-and-block*. Examples are profiling by IP address (a box in front of the server or the server itself admits requests according to a learned demand profile)[3,4, 5]; profiling based on application-level behavior (the server denies access to clients whose request profiles appear deviant [Ranjan et al. 2006; Srivatsa et al. 2006]); and CAPTCHA-based defenses, which preferentially admit humans [von Ahn et al. 2004; Kandula et al. 2005; Morein et al. 2003; Gligor 2003; Google Captcha 2005]. These techniques are powerful because they seek to block or explicitly limit unauthorized users, but their discriminations can err. Indeed, detection-based approaches become increasingly brittle as attackers' mimicry of legitimate clients becomes increasingly convincing (see Section 9.2 for elaboration of this point).

For this reason, our philosophy is to *avoid telling apart good and bad clients*. Instead, we strive for a fair allocation, that is, one in which each client is limited to an equal share of the server. With such an allocation, if 10% of the clients are "bad," then those clients would be limited to 10% of the server's resources, though of course the defense would not "know" that these clients are "bad".[6] One might wonder what happens if 90% of the requesting clients are bad. In this case, a fair allocation still accomplishes something, namely limiting the bad clients to 90% of the resources. However, this "accomplishment" is likely insufficient: unless the server is appropriately overprovisioned, the 10% "slice" that the good clients can claim will not meet their demand. While this fact is unfortunate, observe that if the bad clients look exactly like the good ones but vastly outnumber them, then *no* defense works. (In this case, the only recourse is a proportional allocation together with heavy overprovisioning.)

Unfortunately, in today's Internet, attaining even a fair allocation is impossible. As discussed in Section 3, address hijacking (in which one client appears to be many) and proxies (in which multiple clients appear to be one) prevent the server from knowing how many clients it has or whether a given set of requests originated at one client or many.

As a result, we settle for a *roughly* fair allocation. At a high level, our approach is as follows. The server makes clients reveal how much of some resource they have; examples of suitable resources are CPU cycles, memory cycles, bandwidth, and money. Then, based on this revelation, the server should arrange things so that if a given client has a fraction f of the clientele's total resources, that client can claim up to a fraction f of the server. We call such an allocation a *resource-proportional allocation*. This allocation cannot be "fooled" by the Internet's blurry notion of client identity. Specifically, if multiple clients "pool"

[3]http://mazunetworks.com.
[4]http://www.arbornetworks.com.
[5]http://www.cisco.com.
[6]One might object that this philosophy "treats symptoms", rather than removing the underlying problem. However, eliminating the root of the problem—compromised computers and underground demand for their services—is a separate, long-term effort. Meanwhile, a response to the symptoms is needed today.

their resources, claiming to be one client, or if one client splits its resources over multiple virtual clients, the allocation is unchanged.

Our approach is kin to previous work in which clients must spend some resource to get service [Dwork and Naor 1992; Dwork et al. 2003; Aura et al. 2000; Juels and Brainard 1999; Abadi et al. 2005; Mankins et al. 2001; Wang and Reiter 2007; Back 2002; Feng 2003; Parno et al. 2007; Dean and Stubblefield 2001; Waters et al. 2004; Stavrou et al. 2004]. We call these proposals *resource-based* defenses. Ours falls into this third category. However, the other proposals in this category neither articulate, nor explicitly aim for, the goal of a resource-proportional allocation.[7]

The preceding raises the question: which client resource should the server use? This article investigates *bandwidth*, by which we mean *available upstream bandwidth to the server*. Specifically, when the server is attacked, it encourages all clients to consume their bandwidth (as a way of revealing it); this behavior is what we had promised to justify earlier.

A natural question is, "Why charge clients bandwidth? Why not charge them CPU cycles?" In Section 9.1.1, we give an extended comparison and show that bandwidth has advantages (as well as disadvantages!). For now, we note that one advantage of bandwidth is that it is most likely adversaries' actual constraint. Moreover, many of this article's contributions apply to both currencies; see Section 9.1.1.

1.2 Speak-Up

We present these contributions in the context of *speak-up*, a system that defends against application-level DDoS by charging clients bandwidth for access. We believe that our work [Walfish et al. 2005; Walfish et al. 2006] was the first to investigate this idea, though Sherr et al. [2005] and Gunter et al. [2004] share the same high-level motivation; see Section 9.1.

The central component in speak-up is a server front-end that does the following when the server is oversubscribed: (1) rate-limits requests to the server; (2) encourages clients to send more traffic; and (3) allocates the server in proportion to the bandwidth spent by clients. This article describes several encouragement and proportional allocation mechanisms. Each of them is simple, resists gaming by adversaries, and finds the price of service (in bits) automatically without requiring the front-end and clients to guess the correct price or communicate about it. Moreover, these mechanisms apply to other currencies. The encouragement and proportional allocation mechanism that we implement and evaluate is a *virtual auction*: the front-end causes each client to automatically send a congestion-controlled stream of dummy bytes on a separate payment channel. When the server is ready to process a request, the front-end admits the client that has sent the most bytes.

As a concrete instantiation of speak-up, we implemented the front-end for Web servers. When the protected Web server is overloaded, the front-end gives JavaScript to unmodified Web clients that makes them send large HTTP

[7]An exception is a paper by Parno et al. [2007], which was published after our earlier work [Walfish et al. 2006]; see Section 9.1.

POSTs. These POSTs are the "bandwidth payment." Our main experimental finding is that this implementation meets our goal of allocating the protected server's resources in rough proportion to clients' upload bandwidths.

1.3 How This Article is Organized

The article proceeds in two stages. The first stage is a quick overview. It consists of the general threat and the high-level solution (this section), followed by responses to common questions (Section 2). The second stage follows a particular argument. Here is the argument's outline.

—We give a detailed description of the threat and of two conditions for addressing the threat (Section 3). The first of these conditions is inherent in any defense to this threat.
—We then describe a design goal that, if met, would mitigate the threat—and fully defend against it, under the first condition (Section 4.1).
—We next give a set of designs that, under the second condition, meet the goal (Section 4.2–Section 4.5).
—We describe our implementation and our evaluation of that implementation; our main finding is that the implementation roughly meets the goal (Sections 7–8).
—At this point, having shown that speak-up "meets its spec," we consider whether it is an appropriate choice: we compare speak-up to alternatives and critique it (Section 9). And we reflect on the plausibility of the threat itself and how attackers will respond to speak-up (Section 10).

With respect to this plausibility, one might well wonder how often application-level attacks happen today and whether they are in fact difficult to filter. We answer this question in Section 10: according to anecdote, current application-level attacks happen today, but they are primitive. However, in evaluating the need for speak-up, we believe that the right questions are not about how often the threat *has* happened but rather about whether the threat *could* happen. (And it can.) Simply put, prudence requires proactivity. We need to consider weaknesses before they are exploited.

At the end of the article (Section 11), we depart from this specific threat, in two ways. First, we observe that one may, in practice, be able to combine speak-up with other defenses. Second, we mention other attacks, besides application-level denial-of-service, that could call for speak-up.

2. QUESTIONS

In this section, we answer five nagging questions about speak-up. Appendix A answers many other common questions. Readers with immediate questions are encouraged to turn to this appendix now. While the main text of the article tries to answer many of the questions, consulting the appendix while reading the main text may still be useful.

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—is required. Similarly, the amount of overprovisioning needed at a site defended by speak-up is much less than what a nondefended 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 not under attack. 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 Section 4.1). For a server with spare capacity 90%, the legitimate clientele needs only 1/9th of the aggregate bandwidth of the attacking clients. In Section 10.2, we elaborate on this point and discuss it in the context of today's botnet sizes.

Then couldn't small Web sites, even if defended by speak-up, still be harmed? Yes. There have been reports of large botnets [TechWeb News 2005; Handley 2005; Honeynet Project and Research Alliance 2005; Brown 2006; McLaughlin 2004; Dagon et al. 2006]. If attacked by such a botnet, a speak-up-defended site would need a large clientele or vast overprovisioning to withstand attack fully. However, most botnets are much smaller, as we discuss in Section 10.2. Moreover, as stated in Section 1.1, every defense has this "problem": no defense can work against a huge population of bad clients, if the good and bad clients are indistinguishable.

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 overprovisioned (see, e.g., Fraleigh et al. [2003]), so it could absorb such an increase. (Of course, this overprovisioning could change over time, for example with fiber in homes.) Finally, speak-up's additional traffic is congestion-controlled and will share fairly with other traffic. We address this question more fully in Section 5.

Couldn't speak-up "edge out" other network activity at the user's access link, thereby introducing an opportunity cost? Yes. When triggered, speak-up *may* be a heavy network consumer. Some users will not mind this fact. Others will, and they can avoid speak-up's opportunity cost by leaving the attacked service (see Section 9.1.1 for further discussion of this point).

3. THREAT MODEL AND APPLICABILITY CONDITIONS

The preceding section gave a general picture of speak-up's applicability. We now give a more precise description. We begin with the threat model and then state the conditions that are required for speak-up to be most effective.

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.) As mentioned in Section 1, such an assault is called an application-level attack [Handley 2005]. The clientele of the server is neither predefined (otherwise the server can install filters to permit traffic only from known clients) nor exclusively human (ruling out proof-of-humanity tests [von Ahn et al. 2004; Kandula et al. 2005; Morein et al. 2003; Gligor 2003; Google Captcha 2005; Park et al. 2006]).

Each attacker sends traffic from many hosts; often the attacker uses an army of bots, known as a *botnet*. Because the bots can be distributed all over the world, it is hard to filter the traffic based on the network or geographic origin of the traffic. And the traffic obeys all protocols, so the server has no easy way to tell from a single request that it was issued with ill intent.

Moreover, it may be hard for the server to attribute a collection of requests to the client that sent them. The reason is that the Internet has no robust notion of host identity, by which we mean two things. First, via *address hijacking*, attackers can pretend to have multiple IP addresses. Address hijacking is more than a host simply fudging the source IP address of its packets—a host can actually be reachable at the adopted addresses. We describe the details of this attack in Appendix B; it can be carried out either by bots or by computers that the attacker actually owns. The result is that an abusively heavy client of a site may not be identifiable as such. Second, while address hijacking is of course antisocial, there is socially acceptable Internet behavior with a similar effect, namely deploying NATs (Network Address Translators) and proxies. Whereas address hijacking allows one host to adopt several identities, NATs and proxies cause multiple hosts to share a single identity (thousands of hosts behind a proxy or NAT may share a handful of IP addresses).

Most services handle requests of varying difficulty (e.g., database queries with very different completion times). While servers may not be able to determine the difficulty of a request a priori, our threat model presumes that attackers can send difficult requests intentionally.

We are not considering link attacks. We assume that the server's access links are not flooded; see condition C2 in the following.

The canonical example of a service that is threatened by the attack just described 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). Such sites devote significant computational resources—seconds of CPU time or more—to any client that sends them a request. (Request latencies to Google are of course fractions of a second, but it is highly likely that each request is concurrently handled by tens of CPUs or more.) Other examples are sites that expose a database via a DNS front-end (e.g., the blacklist Spamhaus[8]) and a service like OpenDHT [Rhea et al. 2005], in which clients are invited to

[8]http://www.spamhaus.org.

consume storage anonymously and make requests by Remote Procedure Call (RPC). Beyond these server applications, other vulnerable services include the capability allocators in network architectures such as TVA [Yang et al. 2005] and SIFF [Yaar et al. 2004].[9]

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

C1. 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. This condition is fundamental to any defense to the threat above, in which good and bad clients are indistinguishable: as discussed in Sections 1.1 and 2, if the bad population vastly outnumbers the good population, then *no* defense works.

C2. 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). This condition is one of the main costs of speak-up, relative to other defenses. However, we do not believe that it is insurmountable. First, observe that most Web sites use far less inbound bandwidth than outbound bandwidth (most Web requests are small yet generate big replies).[10] Thus, the inbound request stream to a server could inflate by many multiples before the inbound bandwidth equals the outbound bandwidth. Second, if that headroom is not enough, then servers can satisfy the condition in various ways. Options include a permanent high-bandwidth access link, colocation at a data center, or temporarily acquiring more bandwidth using commercial services (e.g., Prolexic Technologies, Inc. and BT Counterpane). A further option, which we expect to be the most common deployment scenario, is 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 by Agarwal et al. [2003].

Later in this article (Section 10), we reflect on the extent to which this threat is a practical concern and on whether the conditions are reasonable in practice. We also evaluate how speak-up performs when condition C2 isn't met (Section 8.8).

4. DESIGN

Speak-up is motivated by a simple observation about bad clients: they send requests to victimized servers at much higher rates than legitimate clients

[9]Such systems are intended to defend against DoS attacks. Briefly, they work as follows: to gain access to a protected server, clients request tokens, or capabilities, from an allocator. The allocator meters its replies (for example, to match the server's capacity). Then, routers pass traffic only from clients with valid capabilities, thereby protecting the server from overload. In such systems, the capability allocator itself is vulnerable to attack. See Section 9.3 for more detail.

[10]As one datum, consider Wikimedia, the host of `http://www.wikipedia.org`. According to [Weber 2007b], for the 12 months ending in August, 2007, the organization's outbound bandwidth consumption was six times its inbound. And for the month of August, 2007, Wikimedia's outbound consumption was eight times its inbound.

do. (This observation has also been made by many others, including the authors of profiling and detection methods. Indeed, if bad clients weren't sending at higher rates, then, as long as their numbers didn't dominate the number of good clients, modest overprovisioning of the server would address the attack.)

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 simple 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 Web servers receiving single-packet requests, DNS (Domain Name System) front-ends such as those used by content distribution networks or infrastructures like CoDoNS [Ramasubramanian and Sirer 2004], and AFS (Andrew File System) 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 receive only a fraction $\frac{g}{g+B}$ of the server's resources. Assuming $B \gg g$ (if $B \approx g$, then overprovisioning 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 (by assumption). 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 underspecified illustration practical. In the rest of this section, we assume that all requests cause equal server work. We begin with requirements (Section 4.1) and then develop two ways to realize these requirements (Sections 4.2, 4.3).

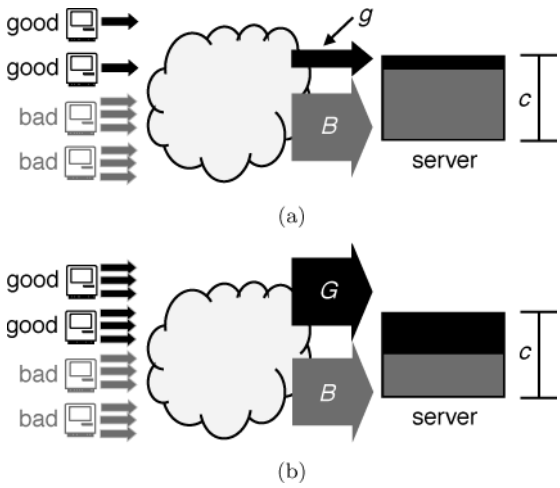

Fig. 1. Illustration of speak-up. The figure depicts an attacked server, $B + g > c$. In (a), the server is not defended. In (b), the good clients send a much higher volume of traffic, thereby capturing much more of the server than before. The good clients' traffic is black, as is the portion of the server that they capture.

We then consider the connections between these approaches as we reflect more generally on the space of design possibilities (Section 4.5). We also consider various attacks (Section 4.4). We revisit our assumptions in Section 5 and describe how speak-up handles heterogeneous requests in Section 6.

4.1 Design Goal and Required Mechanisms

Design Goal. As explained at the beginning of this article, speak-up's principal goal is to allocate resources to competing clients in proportion to their bandwidths:

> Consider a server that can handle c requests per second. If the good clients make g requests per second in aggregate and have aggregate bandwidth of G requests per second to the server, and if the bad clients have 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 overprovisioning 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 that the good clients *can* get must exceed their demand). This expression translates to the *idealized server provisioning requirement*:

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

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 C1 in Section 3), then the required over-provisioning is a factor

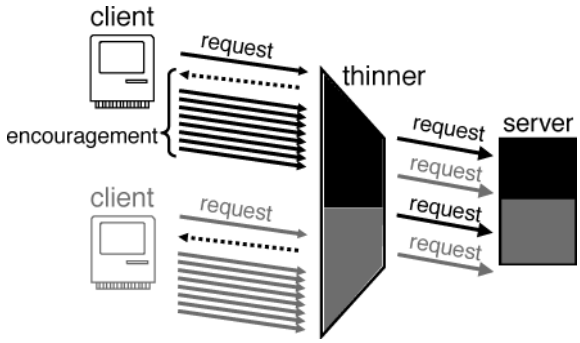

Fig. 2. The components of speak-up. The thinner rate-limits requests to the server. The encouragement signal is depicted by a dashed line from thinner to client (the particular encouragement mechanism is unspecified). In this example, there are two clients, and they send equal amounts of traffic. Thus, the thinner's proportional allocation mechanism (unspecified in this figure) admits requests so that each client claims half of the server.

of two ($c \geq 2g$). In practice, speak-up cannot exactly achieve this ideal because limited cheating is possible. We analyze this effect in Section 4.4.

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 not under attack. Third, given the incoming bandwidths, speak-up needs a *proportional allocation* mechanism to admit clients at rates proportional to their delivered bandwidths.

Under speak-up, these mechanisms are implemented by a front-end to the server, called the *thinner*. As depicted in Figure 2, the thinner implements encouragement and controls which requests the server sees. Encouragement and proportional allocation can each take several forms, as we will see in the two variations of speak-up in Section 4.2, 4.3. And we will see in Section 4.5 that the choices of encouragement mechanism and proportional allocation mechanism are orthogonal.

Before presenting these specifics, 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*) causes good clients to send significantly more traffic—while still obeying congestion control.

Bad Clients and Congestion Control. Of course, bad clients may not obey congestion control, even though speak-up's protocols call for it. The result of such disobedience is as follows.

First, in the rest of this section (Sections 4.2–4.5), we assume that no bottleneck network link is shared by multiple clients. In this case, a bad client's advantage from disobeying congestion control is minor (as discussed in Section 4.4).

Second, in Section 5, we consider shared bottleneck links. In this case, if a bad client is behind such a link but does not obey congestion control, that client is conducting a link attack, which is outside speak-up's scope (see Section 3). If a bad client is behind such a link and does obey congestion control, the bad client can adversely affect a good client of speak-up; however, regardless of whether speak-up is in effect, if a good and bad client share a bottleneck link, the good client loses. We discuss this issue further in Section 5 and experimentally evaluate this case in Section 8.6.

Finally, if the thinner's network link is congested, bad clients are again advantaged—regardless of whether they obey congestion control. We experimentally evaluate a congested thinner in Section 8.8.

4.2 Aggressive Retries and Random Drops

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 that clients are not identifiable: with address hijacking, discussed in Section 3, one client may claim to be several, and with NATs and proxies, 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.

But why doesn't the continuous stream of bytes require such identification (for example, to attribute bytes to clients)? The reason is as follows. First, asking each client for a continuous stream forces each client to reveal its bandwidth (by consuming it). Second, with the bandwidth so revealed, speak-up can do its job, which, per the design goal in Section 4.1, is to allocate service to a group

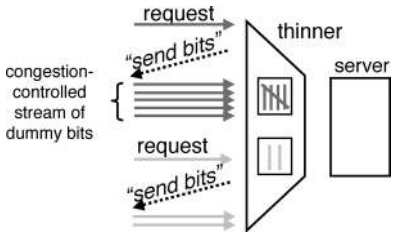

Fig. 3. Speak-up with an explicit payment channel. For each request that arrives when the server is busy, the thinner asks the requesting client to send dummy bits. Imagine that an auction is about to happen. The dark gray request will win the auction because it has five units of payment associated with it, compared to only two units for the light gray request.

in proportion to the bandwidth available to that group. Specifically, if a client claims to be several clients, or several clients can pool their resources to appear as one, speak-up continues to meet its goal: either a larger number of manufactured clients will individually get less service, or one "pooled" client will get the service owed to the component clients. To put the foregoing differently, taxing clients—charging them for service—is easier than identifying them (ironically).

Indeed, speak-up is a currency-based scheme (as we said earlier), and the price for access is the average 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, as can be seen from the expressions for r, derived below.

This approach fulfills the design goal in Section 4.1, 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, and $r = B/(c - g)$ (as we show immediately below). Note that in both cases r changes with the attack size, B.

To see that $r = B/(c - g)$ in the second case, observe that the "bad load" that actually reaches the server reduces from B, the attackers' full budget, to Bp. Thus, the thinner's dropping, combined with the fact that good clients retry their "good load" of g until getting through, results in the equation $g + Bp = c$, which implies $r = 1/p = B/(c - g)$.

4.3 Explicit Payment Channel and Virtual Auction

We now describe another encouragement mechanism and another proportional allocation mechanism; we use these mechanisms in our implementation and evaluation. They are depicted in Figure 3. For encouragement, the thinner does

the following. When the server is oversubscribed, the thinner asks a requesting client to open a separate *payment channel*. The client then sends a congestion-controlled stream of bits on this channel. (Conceptually, the client is padding dummy bytes to its request.) We call a client that is sending bits a *contending* client; the thinner tracks how many bits each contending client sends.

Note that these contending clients may be manufactured identities, or they may be pooled "super-clients"; as mentioned in Section 4.2, these cases do not thwart speak-up. More specifically, in our implementation, the client associates its bandwidth payment with its request via a "request id" field *that clients are permitted to forge*. Such forgery amounts to contributing bandwidth to another client or splitting bandwidth over two virtual clients, both of which are behaviors that the design *assumes* of clients, and neither of which detracts from speak-up's design goal of a bandwidth-proportional allocation; see page 3, Section 4.1, Section 4.2, and Section 4.4.

The proportional allocation mechanism is as follows. 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 bits, and it terminates the corresponding payment channel.

As with the version in Section 4.2, the price here emerges naturally. Here, it is expressed in bits 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 bits (not requests) per second and assume that the good and bad clients are "spending everything," so $B + G$ bits per second enter the thinner. Since auctions happen every $1/c$ seconds on average, the average price is $\frac{B+G}{c}$ bits per request.

However, we cannot claim, as in Section 4.2, 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. In the next section, we show that the auction can be gamed but not too badly, so all clients do in fact see prices that are close to the average.

4.4 Cheating and the Virtual Auction

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

PROOF. Consider a client, X, that transmits an α 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, recalling that α is what we called the *fraction* of the total spent by X, we get

$$\alpha \leq \frac{2}{(t/k + 1)}.$$

It follows that

$$\frac{k}{t} \geq \frac{\alpha}{2 - \alpha} \geq \frac{\alpha}{2},$$

that is, X receives at least an $\alpha/2$ 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 Theorem 4.1 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. To relax this assumption, the theorem can be extended as follows: for service times that fluctuate within a bounded range

$[(1-\delta)/c\,,(1+\delta)/c\,]$, X receives at least a $(1-2\delta)\alpha/2$ 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 bits" 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 Section 7), there is a quiescent period between POSTs (equal to one RTT 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 bits 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 $\alpha/2$ service for α bandwidth). This advantage arises because the proof lets the adversary control exactly when its bits 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 two over the ideal from Section 4.1, and this provisioning is still far less than what would be required to absorb the attack without speak-up.

To see that the required provisioning increases by a factor of two, observe that the theorem says that good clients can get service of up to $c\alpha/2 = \frac{Gc}{2(G+B)}$ requests per second. Yet good clients need service of g requests per second. Thus, the required provisioning, which we denote c_{req}, must satisfy $\frac{Gc_{req}}{2(G+B)} \geq g$. This inequality yields $c_{req} \geq 2c_{id}$.

In Section 8.4, 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—affect the required provisioning above the ideal.

4.5 Design Space

We now reflect on the possible designs for speak-up and then discuss how we chose which one to implement and evaluate.

4.5.1 *Axes.* We have so far presented two designs: "aggressive retries and random drops" (Section 4.2) and "payment channel and virtual auction" (Section 4.3). These designs are drawn from a larger space, in which there are two orthogonal axes that correspond to the required mechanisms from Section 4.1:

A1 **Encouragement method**:	A2 **Allocation mechanism**:
—Aggressive retries	—Random drops
—Payment channel	—Virtual auction

Thus, it is possible to imagine two other designs. We discuss them now.

4.5.2 *"Retries and Virtual Auction."* In this design, clients send repeated retries in-band on a congestion-controlled stream. The thinner conducts a periodic auction, selecting as the winner the request with the most retries (rather than the most bits) up to that point. We can apply Theorem 4.1 to this design, as follows. The theorem describes clients' bandwidths in terms of a made-up unit (the "dollar"), so we need only take this unit to be retries between auctions, rather than bits between auctions.

4.5.3 *Payment Channel and Random Drops."* In this design, clients pay bits out of band. As in the "virtual auction" designs, the thinner divides time into service intervals (i.e., time lengths of $1/c$ seconds), making an admission decision at the end of each interval. In this design, however, the intervals are independent. For a given interval, the thinner records how many bits clients have sent in that interval. At the end of an interval, the thinner chooses randomly. Specifically, a request that has sent a fraction f of the total bits in that interval is admitted by the thinner with probability f. To show that this design achieves our goal, we use the following theorem; like Theorem 4.1, it relies on the assumption that requests are served with perfect regularity.

THEOREM 4.2. *Under this design, any client that continuously delivers a fraction α of the average bandwidth received by the thinner gets at least a fraction α of service, in expectation, regardless of how the other clients time or divide up their bandwidth.*

PROOF. As in Theorem 4.1, consider a single client, X, and again assume that X delivers a *dollar* between service intervals. We will examine what happens over t time intervals. Over this period, X delivers t dollars of bandwidth. We are given that all of the clients together deliver t/α dollars over this period, so the other clients deliver $t/\alpha - t$ dollars.

Now, consider each of the i intervals. In each interval, the service that X expects is the same as the probability that it is admitted, which is $1/(1+b_i)$, where b_i is the bandwidth delivered by the other clients in interval i. By linearity of

expectation, the total expected service received by X is

$$\sum_{i=1}^{t} \frac{1}{1+b_i} \quad \text{subject to} \quad \sum_{i=1}^{t} b_i = \frac{t}{\alpha} - t.$$

The minimum—which is the worst case for X—happens when the b_i are equal to each other, that is, $b_i = 1/\alpha - 1$. In that case, the expected service received by X is

$$\sum_{i=1}^{t} \frac{1}{1+1/\alpha-1} = \alpha t,$$

so X can expect at least an α fraction of the total service. □

An advantage of this approach is that the thinner need not keep track of how many bits have been sent on behalf of each request, as we now explain. (Of course, depending on the scenario, the thinner may still need per-request state, such as congestion control state or other connection state.) We can regard the thinner's tasks as (a) receiving a stream of bits in each interval (each packet brings a set of bits on behalf of a request); (b) at the end of the interval, choosing a bit uniformly at random; and (c) admitting the request on whose behalf the "winning bit" arrived. These tasks correspond to admitting a request with probability proportional to the number of bits that were sent on behalf of that request in the given interval.

To implement these tasks, the thinner can use *reservoir sampling* [Knuth 1998] with a reservoir of one bit (and its associated request). Reservoir sampling takes as input a stream of unknown size and flips an appropriately weighted coin to make a "keep-or-drop" decision for each item in the stream (a "keep" decision evicts a previously kept item). The weights on the coins ensure that, when the stream ends, the algorithm will have chosen a uniformly random sample of size k (in our case, $k = 1$). A further refinement avoids making a decision for each item (or bit, in our context): once the algorithm keeps an item, it chooses a random value representing the *next* item to admit; it can then discard all intermediate items with a clear conscience [Vitter 1985].

4.5.4 *Comparing the Possibilities.* Before we actually compare the alternatives, observe that one of the designs is underspecified: in the description of "aggressive retries and random drops" in Section 4.2, we did not say how to set p, the drop probability. However, the design and theorem in Section 4.5.3 suggest one way to do so: the thinner divides up time and selects one client to "win" each interval (rather than trying to apply a drop probability to each retry independently such that the total rate of admitted requests is c). With this method of setting p, every design in this space shares the same high-level structure: clients pipeline bits or requests, and the thinner selects a client once every $1/c$ seconds. The designs are thus directly comparable.

The differences among the designs are as follows. Axis A1 is primarily an implementation distinction, and which choice is appropriate depends on the protected application and on how speak-up fits into the communication protocol between clients and servers.

Axis A2 is more substantive. Here, we have a classic trade-off between random and deterministic algorithms. The "virtual auction" is gameable in a limited way, but clients' waiting times are bounded: once a client has paid enough, it is served. "Random drops" is the opposite: it is not at all gameable, but our claims about it apply only to long-term averages. At short time scales, there will be significant variation in the server's allocation and thus in waiting times. In particular, a typical coupon-collector analysis shows that if there are n equivalent clients that continually make requests, some of the clients will have to wait an expected $O(n \log n)$ intervals to get service. (Regard each interval as picking a client uniformly at random.) Another difference between the two options is that one can implement "random drops" with less state at the thinner (by using reservoir sampling, as already described). Which choice on axis 2 is appropriate depends on one's goals and taste.

Rationale for our choices. For our prototype (Section 7), we chose the payment channel over in-band retries for reasons related to how JavaScript drives Web browsers. We chose the virtual auction over random drops because we wanted to avoid variance.

Other designs. One might wonder whether the design space is larger than these four possibilities, which share a similar structure. Indeed, we used to be enamored of a different structure, namely the version of "random drops" described at the beginning of Section 4.2. The charm of that version was that its thinner was stateless. However, we ultimately rejected that approach because, as described in Section 4.2, clients need to keep their pipes to the thinner full (otherwise, recall, bad clients could manufacture a bandwidth advantage). This requirement implies that the thinner must maintain congestion control state for each client, ending the dream of a stateless thinner.

5. 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 first assumption, but we address them 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.

5.1 Speak-Up's Effect on the Network

No flow between a good client and a thinner individually exhibits antisocial behavior. In our implementation (see Section 7), each payment channel comprises a series of HTTP POSTs and thus inherits TCP's congestion control. For UDP applications, the payment channel could use the congestion manager [Balakrishnan et al. 1999] or DCCP [Kohler et al. 2006]. (Bad clients can refuse to control congestion, but this behavior is a link attack, which speak-up does not defend against; see Section 3.) However, individually courteous flows do not automatically excuse the larger rudeness of increased traffic levels, and we must ask whether the network can handle this increase.

To answer this question, we give two sketchy arguments suggesting that speak-up would not increase total traffic much, 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 [Fraleigh et al. 2003]. 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 servers is under attack at any given 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 [Fraleigh et al. 2003] suggest that these links operate at low utilization. And, while the core cannot handle every client transmitting maximally (as argued by Vasudevan et al. [2006]), 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 (as will any heavy user of the network), an effect that we explore experimentally in Section 8.7.

5.2 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 (Section 4.4), 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,[11] 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 Section 8.6.

5.3 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 C2 in Section 3). It also needs enough processing capacity to handle the dummy bits. (Meeting these requirements is far easier than provisioning the

[11] One might argue that, if the server were to charge CPU cycles instead, then an analogous outcome would not happen. However, as we observe in Section 9.1.1, a CPU can also be a shared resource.

server to handle the full attack: unlike the server, the thinner does not do much per-request processing.) We now argue that meeting these requirements is plausible.

One study [Sekar et al. 2006] of observed DoS attacks found that the 95th percentile of attack size was in the low hundreds of Mbits/s (see Figure 15), which agrees with other anecdotes [Thomas 2005]. 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., Prolexic Technologies, Inc., BT Counterpane), 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 C2 in Section 3). 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 Section 8.1. A production solution would presumably do much better.

5.4 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 are not 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 Section 4 still hold: they are in terms of the bandwidth *available* to both populations, not the bandwidth that they actually use today.

6. HETEROGENEOUS REQUESTS

We now generalize the design to handle the more realistic case in which the requests are unequal. There are two possibilities: either the thinner can tell the difficulty of a request in advance, or it cannot. In the first case, the design is straightforward: the thinner simply scales a given bandwidth payment by the difficulty of the associated request (causing clients to pay more for harder requests).

In the remainder of this section, we address the second case, making the worst-case assumption that although the thinner does not know the difficulty of requests in advance, attackers do, as given by the threat model in Section 3. 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 do not 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 view a request as comprising equal-sized *chunks* that each 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 Section 4.3), 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.

In general, the approach described in this section could apply to other defenses as well (though, to our knowledge, no one has proposed it). For example, a profiler could allocate quanta based on clients' historical demand rather than on how many bits clients have paid.

7. IMPLEMENTATION

We implemented a prototype thinner in C++ as an OKWS [Krohn 2004] Web service using the SFS toolkit [Mazières 2001]. It runs on Linux 2.6. 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.

The thinner is designed to be easy to deploy. It is a Web front-end that is intended to run on a separate machine "in front" of the protected Web server

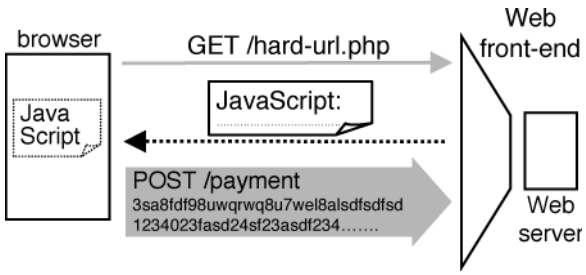

Fig. 4. Implementation of the payment channel. When the server is busy, the thinner, implemented as a Web front-end, sends JavaScript to clients that causes them to send large HTTP POSTs. The thinner ends a POST if and when the client wins the auction.

(which we will call just the *server* in the remainder of this section). Moreover, there are not many configuration parameters. They are:

—the *capacity* of the protected server, expressed in requests per second.
—a *list of URLs and regular expressions* that correspond to "hard requests." Each URL and regular expression is associated with a *difficulty level*. This difficulty level is relative to the capacity. For example, if the server's capacity is 100 requests per second, and if the thinner is configured such that a given URL has difficulty 2, then the thinner assumes that for the server to handle that request takes an average of .02 seconds.
—the *name or address* of the server.
—a custom *"please wait" screen* that humans will see while the server is working and while their browser is paying bits. Existing computationally intensive Web sites already use such screens.

When the thinner gets a request, it first checks whether that request is on the list of hard requests. If not, it passes the request directly to the server and feeds the response back to the client on behalf of the server.

On the other hand, if the Web client requested a "hard" URL, the thinner replies immediately with the "please wait" screen. If there are no other connections to the thinner (i.e., if the server is not oversubscribed), then the thinner submits the client's request to the server. After the server processes the request and replies to the thinner, the thinner returns to the client (1) JavaScript that wipes the "please wait" screen and (2) the contents of the server's reply.

If, however, other clients are communicating with the thinner (i.e., if the server is oversubscribed), the thinner adds JavaScript after the "please wait" HTML. As depicted in Figure 4, this JavaScript causes the client's browser to dynamically construct, and then submit, a one-megabyte HTTP POST containing random bytes. (One megabyte reflects some browsers' limits on POSTs.) This POST is the client's bandwidth payment (Section 4.3). If, while sending these dummy bytes, the client wins an auction (we say below when auctions happen), the thinner terminates the POST and submits the client's request to the server, as above. And, as above, the server then replies to the thinner, the thinner wipes the "please wait" screen, etc.

If the client completes the POST without winning an auction, then the thinner returns JavaScript that causes the browser to send another POST, and the process described in the previous paragraph repeats.

The thinner correlates the client's payments with its request via a "request id" field in all HTTP requests. This field can be forged by a client. As mentioned in Section 4.3, our design is robust to such forgery and in fact assumes that such forgery is happening.

Auctions. The thinner holds auctions (and demands bandwidth payment) whenever it has more than one connection open (this state corresponds to oversubscription of the server). The server does not tell the thinner whether it is free. Rather, the thinner uses the configuration parameters (specifically, the server's capacity and the difficulty of requests) to meter requests to the server in the obvious way: if we assume for a moment that all requests are of unit difficulty, then the thinner holds an auction every $1/c$ seconds. Backing off of this assumption, if a request of difficulty level d has just been admitted to the server, then the thinner will hold the next auction d/c seconds later. To handle difficult requests fairly, the thinner scales clients' payments by the difficulty level, and the auction winner is based on the scaled payments.

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.

8. 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 Section 8.2 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 Section 4.1. In Section 8.3, we measure speak-up's latency and byte cost. In Section 8.4, we ask how much bad clients can "cheat" speak-up to get more than a bandwidth-proportional share of the server. Section 8.5 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 (Section 8.6) and with non-speak-up traffic (Section 8.7). Finally, we measure how speak-up performs when the thinner's bandwidth is under-provisioned (Section 8.8); that is, we measure the effect of not meeting condition C2 in Section 3. Table I summarizes our results.

8.1 Setup and Method

All of the experiments described here ran on the Emulab testbed.[12] The clients run a custom Python Web client and connect to the prototype thinner in various

[12]http://www.emulab.net.

Table I. Summary of Main Evaluation Results

Our thinner implementation allocates a protected server in rough proportion to clients' bandwidths.	Section 8.2, Section 8.5
In our experiments, the server needs to provision only 37% beyond the bandwidth-proportional ideal to serve 99.98% of the good requests.	Section 8.3, Section 8.4
Our unoptimized thinner implementation can sink 1.5 Gbits/s of uploaded "payment traffic".	Section 8.1
On a bottleneck link, speak-up traffic can crowd out other speak-up traffic and non-speak-up traffic.	Section 8.6, Section 8.7
When the thinner has less bandwidth than required (i.e., when condition C2 from Section 3 is not met), speak-up does not achieve a bandwidth-proportional allocation but still yields a better allocation than having no defense.	Section 8.8

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.

The protected server is an Apache Web server that runs on the same host as the thinner. Recall that a thinner is configured with a list of "hard requests" (Section 7). In our experiments, the "hard request" is a URL, U. When the Web server receives HTTP GET requests for U (passed by the thinner) it invokes a simple PHP script that sleeps for $1/c$ seconds and then returns a short text file. c is the server capacity that we are modeling and varies depending on the experiment. The thinner sends the server requests for U not more often than once every $1/c$ seconds. If a request arrives while the server is still "processing" (really, sleeping on behalf of) a previous one, the thinner replies with JavaScript that makes the client issue a one megabyte HTTP POST—the payment bytes (see Section 7).

All experiments run for 600 seconds. Each client runs on a separate Emulab host and generates *requests* for U. All requests are identical. Each client's requests are driven by a Poisson process of rate λ requests/s.[13] 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.

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.

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

[13]We chose Poisson processes for convenience; our theorems in Section 4 apply to such processes and any other client behavior.

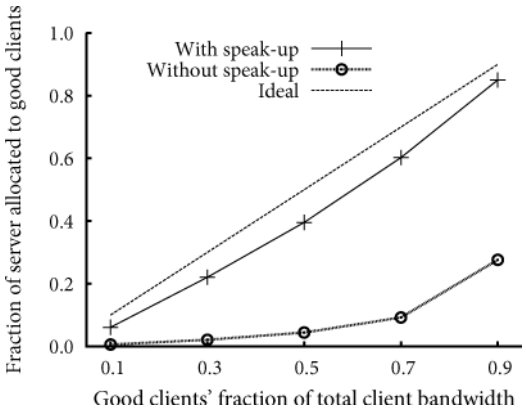

Fig. 5. Server allocation when $c = 100$ requests/s as a function of $\frac{G}{G+B}$. The measured results for speak-up are close to the ideal line. Without speak-up, bad clients sending at $\lambda = 40$ requests/s and $w = 20$ capture much more of the server.

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 Section 4. By understanding this difference, one 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 recent attacks are roughly this size; see Section 5.3 and Section 10.2. The capacity also depends on how many concurrent clients the thinner supports; the limit here is only the RAM for each connection (see Section 7).

8.2 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 5 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

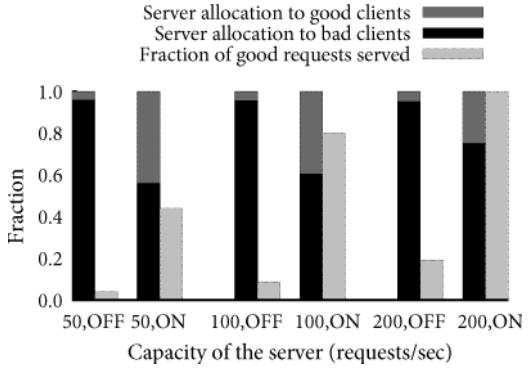

Fig. 6. Server allocation to good and bad clients, and the fraction of good requests that are served, without ("OFF") and with ("ON") speak-up. c varies, and $G = B = 50$ Mbits/s. For $c = 50, 100$, the allocation is roughly proportional to the aggregate bandwidths, and for $c = 200$, all good requests are served.

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 Section 4.4 and further quantify it in Section 8.3 and Section 8.4.

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 Section 4.1 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 Section 4.1, $c_{id} = g(1 + \frac{B}{G}) = 2g = 2 \cdot 25 \cdot \lambda = 100$), and we experiment with $c = 50, 100, 200$ requests/s.

Figure 6 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. (The bad clients get a larger share of the server for $c = 200$ because they capture the excess capacity after the good requests have been served.) Again, one can see that the allocation under speak-up does not exactly match the ideal: from Figure 6, when speak-up is enabled and $c = c_{id} = 100$, the good demand is not fully satisfied.

8.3 Latency and Byte Cost

We now explore speak-up's byte cost and a pessimistic estimate of its latency cost for the same set of experiments (c varies, 50 clients, $G = B = 50$ Mbits/s).

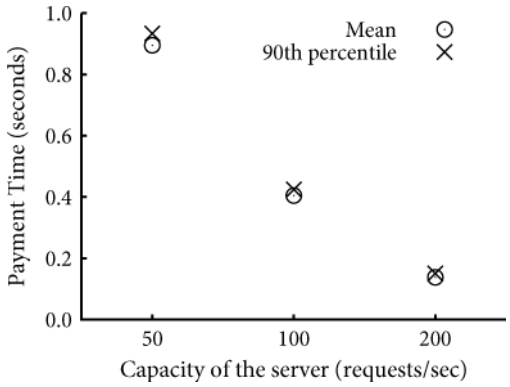

Fig. 7. Mean time to upload dummy bytes for good requests that receive service. c varies, and $G = B = 50$ Mbits/s. When the server is not overloaded ($c = 200$), speak-up introduces little latency.

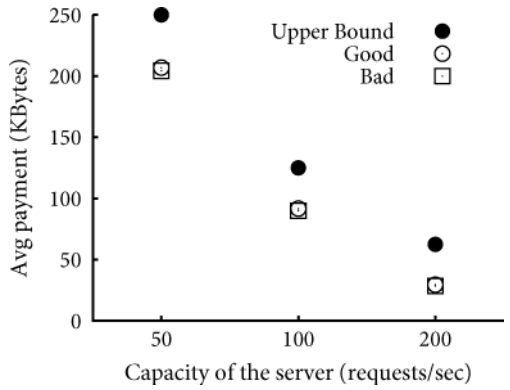

Fig. 8. Average number of bytes sent on the payment channel—the "price"—for served requests. c varies, and $G = B = 50$ Mbits/s. When the server is overloaded ($c = 50, 100$), the price is close to the upper bound, $(G + B)/c$; see the text for why they are not equal.

For the pessimistic latency cost, we measure the length of time that clients spend uploading dummy bytes, as seen at the client. Figure 7 shows the averages and 90th percentiles of these measurements for the served good requests. The reasons that this measurement is a pessimistic reflection of speak-up's true latency cost are as follows. First, for the good clients, speak-up decreases average latency (because speak-up serves more good requests). Second, even calling this measurement the *per-request* latency cost is pessimistic because that view unrealistically implies that, without speak-up, the "lucky" requests (the ones that receive service) never have to wait. Third, any other resource-based defense would also introduce some latency.

For the byte cost, we measure the number of bytes uploaded for served requests—the "price"—as recorded by the thinner. Figure 8 shows the average of this measurement for good and bad clients and also plots the theoretical average price, $(G + B)/c$, from Section 4.3, which is labeled Upper Bound.

We make two observations about this data. The first is that when the server is under-provisioned, good clients pay slightly more for service than bad ones. The reason is as follows. *All* contending clients tend to overpay: the client that will win the next auction continues to pay until the auction happens rather than stopping after it has paid enough to win. And since good clients pay at a faster rate *per request*, they tend to overshoot the "true" price (the second-highest bid) more than the bad clients do. Note, however, that the overpayment by any client is bounded by $\frac{1}{c}\cdot$ (bandwidth of a client) because a client can overpay for at most the time between two auctions.

The second observation is that the actual price is lower than the theoretical one. The reason is that clients do not consume all of their bandwidth. We now explain why they do not, considering the different values of c in turn.

For $c = 50$, each good client spends an average of 1.46 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 issues a request and when the thinner replies, asking for payment. This period is 0.22 seconds on average, owing mostly to a long backlog at the thinner of requests and payment bytes (but a little to round-trip latency). When not in a quiescent period, a good client consumes most of its access link, delivering 1.85 Mbits/s on average, inferred by dividing the average good client payment (Figure 8) by the average time spent paying (Figure 7). Bad clients, in contrast, keep multiple requests outstanding so do not have "downtime." For $c = 50$, they spend an average of 1.84 Mbits/s.

The $c = 100$ case is similar to $c = 50$.

We now consider $c = 200$. Recall that the upper bound on price of $(G + B)/c$ is met only if all clients actually pay all of their bandwidth. However, if the server is over-provisioned and all of the good clients' requests are served, as happens for $c = 200$, then the good clients do not pay the maximum that they are able. For each request, a good client pays enough to get service, and then goes away, until the next request. This behavior causes a lower "going rate" for access than is given by the upper bound.

8.4 Empirical Adversarial Advantage

As just discussed, bad clients 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 Section 8.2.

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 at $c = 137$, 99.98% of the good demand is satisfed and that at $c = 140$, all but one of the good clients' 30,157 requests is served. $c = 137$ is 37% 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.

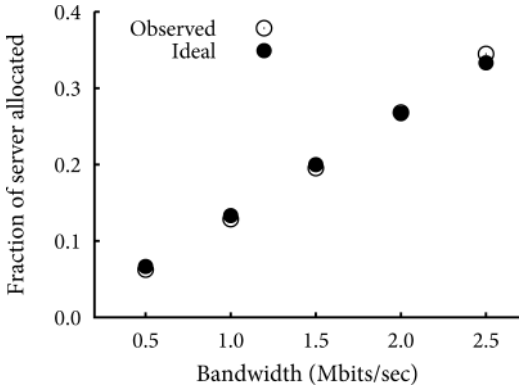

Fig. 9. Heterogeneous client bandwidth experiments with 50 LAN clients, all good. The fraction of the server ($c = 10$ requests/s) allocated to the ten clients in category i, with bandwidth $0.5 \cdot i$ Mbits/s, is close to the ideal proportional allocation.

We now revisit that model. First, we chose $w = 20$ arbitrarily. It might be true that with a smaller or larger value for w, the bad clients could capture more of the server. Second, bad clients do not "optimize." As one example, in the $c = 50$ experiment, the average time between when the thinner returns JavaScript to a bad client and when the thinner actually gets bits from that client is roughly two full seconds. During those two seconds, the bad client is effectively paying for $w - 1$ (or fewer requests) rather than w requests, so perhaps bad clients are not realizing their full adversarial advantage. Indeed, one could imagine a bad client setting w adaptively or concentrating bandwidth on particular requests. Nevertheless, the analysis in Section 4.4 shows that bad clients cannot do much better than the naïve behavior that we model.

8.5 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 9 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, a client will have sent the full price—that is, the requisite number of bytes to win the virtual auction—before TCP has "ramped up" to fill the client's pipe. In these cases, clients with longer RTTs will take longer to pay. Second, and more importantly, each request has at least one associated quiescent period (see Section 8.1 and Section 8.3), the length of which depends on RTT. In contrast, bad clients have multiple requests outstanding so do not have "downtime" and will not be much affected by their RTT to the thinner.

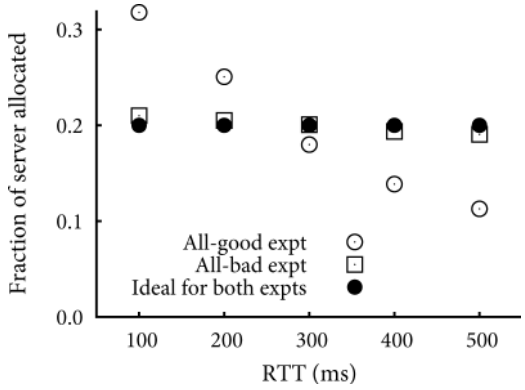

Fig. 10. Two sets of heterogeneous client RTT experiments with 50 LAN clients, all good or all bad. The fraction of the server ($c = 10$ requests/s) captured by the 10 clients in category i, with RTT $100 \cdot i$ ms, varies for good clients. In contrast, bad clients' RTTs do not matter because they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

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 10 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.

8.6 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 Section 5.2). We quantify this observation with an experiment that uses the following topology, depicted in Figure 11: 30 clients, each with a bandwidth of 2 Mbits/s, connect to the thinner through a common link, l. The capacity of l is 20 Mbits/s. l is a bottleneck because the clients behind l can generate 60 Mbits/s. Also, 5 good and 5 bad clients, each with a bandwidth of 2 Mbits/s, connect to the thinner directly through a LAN. The server's capacity is $c = 30$ requests/s. We vary the number of good and bad clients behind l, measuring three cases: first, 5 good and 25 bad clients; second, 15 good and 15 bad clients; and third, 25 good and 5 bad clients.

Based on the topology, the clients behind l should capture half of the server's capacity. In fact, they capture slightly less than half: in the first case, they capture 43.8%; in the second, 47.7%; and in the third, 47.1%.

Next, we measure the allocation of the server to the good and bad clients behind l. We also measure, of the good requests that originate behind l, what fraction receive service. Figure 12 depicts these measurements and compares them to the bandwidth-proportional ideals. The ideal for the first measurement is given simply by the fraction of good and bad clients behind l. The ideal for the

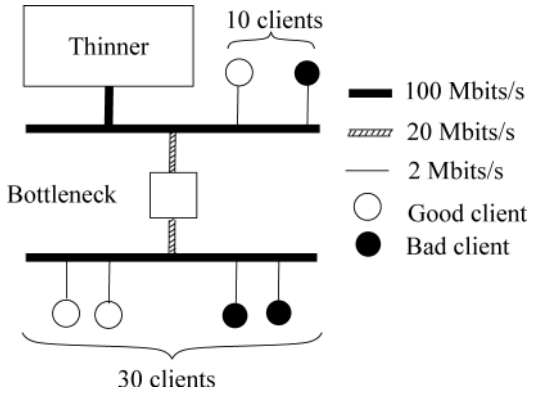

Fig. 11. Network topology used to measure the impact of good and bad clients sharing a bottleneck link (Section 8.6).

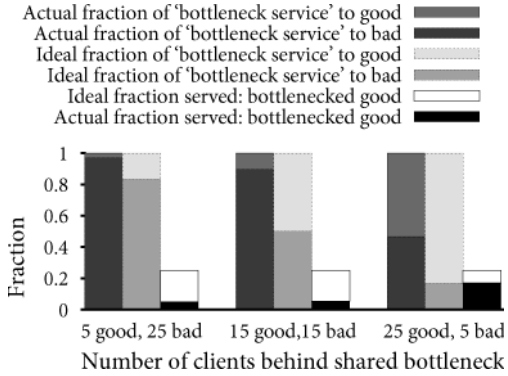

Fig. 12. Server allocation when good and bad clients share a bottleneck link, l. "Bottleneck service" refers to the portion of the server captured by all of the clients behind l. The actual breakdown of this portion (left bar) is worse for the good clients than the bandwidth-proportional allocation (middle bar) because bad clients "hog" l. The right bar further quantifies this effect.

second measurement, f_{id}, is 0.25, and it is calculated as follows. Let G_l be the ideal bandwidth available to the good clients behind l. $G_l = 2\frac{20}{60}n$, where n is the number of good clients behind l. The fraction $\frac{20}{60}$ reflects the fact that in the ideal case, the bottleneck restricts every client equally. Further, let $g_l = n\lambda$ be the rate at which the good clients behind l issue requests. Of the good requests that originate behind l, the ideal fraction that would be served, f_{id}, is the bandwidth-proportional server piece for the good clients behind l divided by those clients' demand:

$$f_{id} = \frac{\frac{G_l}{G+B}c}{g_l} = \frac{\frac{G_l}{40}30}{g_l} = \frac{2\frac{20}{60}n30}{40n\lambda} = \frac{2\frac{20}{60}n30}{80n} = 0.25.$$

The figure shows that the good clients behind l are heavily penalized. The reason is that each bad client keeps multiple connections outstanding so captures

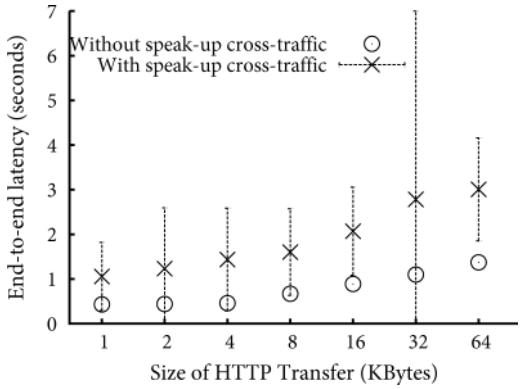

Fig. 13. Effect on an HTTP client of sharing a bottleneck link with speak-up clients. Graph shows means of end-to-end HTTP download latencies with and without cross-traffic from speak-up, for various HTTP transfer sizes (which are shown on a log scale). Graph also shows standard deviations for the former measurements (the standard deviations for the latter are less than 1.5% of the means). Cross-traffic from speak-up has a significant effect on end-to-end HTTP download latency.

much more of the bottleneck link, l, than each good client. This effect was hypothesized in Section 5.2.

8.7 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 uploading dummy bits. 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 13 shows the mean download latency for various file sizes, with and without the speak-up traffic. The figure also shows standard deviations for the former set of measurements. For the latter set, the standard deviations are less than 1.5% of the means.

There is significant "collateral damage" to "innocently bystanding" Web transfers here: download times inflate between 2 and 3.2× for the various transfer sizes. 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, it will not have this effect on every link.

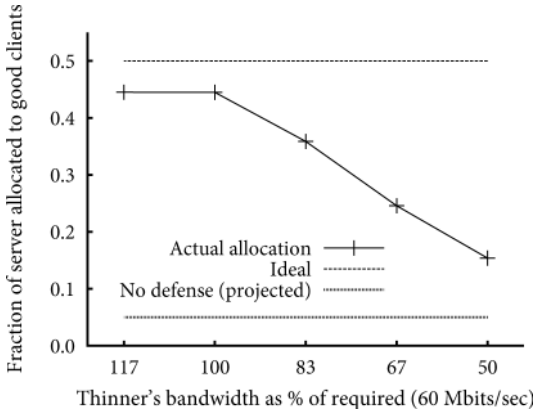

Fig. 14. Server allocation as a function of the thinner's bandwidth provisioning. $G = B = 30$ Mbits/s, and $c = 30$ requests/s. Graph depicts the actual allocation that we measured under speak-up, as well as the ideal allocation and our prediction of the allocation that would result if no defense were deployed. As the thinner becomes more under-provisioned, bad clients capture an increasingly disproportionate fraction of the server. However, even in these cases, speak-up is far better than having no defense.

8.8 Underprovisioned Thinner

We now explore how speak-up performs when condition C2 from Section 3 is not met. That is, we measure what happens when the thinner's incoming bandwidth is less than the combined bandwidth of its current clients. In this case, we expect bad clients to claim a disproportionate share of the server. The reason is as follows. When the thinner's access link is overloaded, the thinner will not "hear" all of the incoming requests. Thus, not all of the incoming requests will receive timely encouragement. Because good clients make fewer requests and keep fewer requests outstanding, a good client, relative to a bad client, has a higher probability of having no "encouraged" requests outstanding. Thus, on average, a good client is quiescent more often than a bad client, meaning that a good client pays fewer bits and hence gets less service.

To measure this effect, we perform an experiment with the following topology: 15 good and 15 bad clients, each with a bandwidth of 2 Mbits/s, connect to the thinner over a LAN. Observe that the thinner's required bandwidth provisioning is 60 Mbits/s. Our experiments vary the thinner's actual access bandwidth from 30 Mbits/s, representing a thinner that is underprovisioned by 50%, to 70 Mbits/s, representing a thinner that is amply provisioned. For each case, we measure the fraction of the server that goes to good and bad clients.

Figure 14 depicts the results, along with the ideal allocation and the allocation that we project would result if no defense were deployed. This latter allocation is the ratio of the two populations' request rates: $\lambda = 2$ for a good client and $\lambda = 40$ for a bad client (see Section 8.1).

As the thinner becomes increasingly under-provisioned, bad clients capture increasingly more of the server, which is the effect that we hypothesized. Nevertheless, even in this case, speak-up is far better than nothing.

9. SPEAK-UP COMPARED & CRITIQUED

Having shown that speak-up roughly "meets its spec," we now compare it to other defenses against application-level DDoS attacks. (For pointers to the broad literature on other denial-of-service attacks and defenses, in particular link attacks, see the survey by Mirkovic and Reiher [2004] and the bibliographies in Yang et al. [2005], Kandula et al. [2005], and Morein et al. [2003].) Some of the defenses that we discuss in the following have been proposed; others are hypothetical but represent natural alternatives. As the discussion proceeds, we will also critique speak-up. Some of these critiques are specific to speak-up; some apply to speak-up's general category, to which we turn now.

9.1 Resource-Based Defenses

This category was pioneered by Dwork and Naor [1992], who suggested, as a spam defense, having receivers ask senders for the solutions to computationally intensive puzzles, in effect charging CPU cycles to send email. (Back later proposed a similar idea [2002].) Since then, others have done work in the same spirit, using *proof-of-work* (as such schemes are known) to defend against denial-of-service attacks [Dwork et al. 2003; Aura et al. 2000; Juels and Brainard 1999; Feng 2003; Parno et al. 2007; Dean and Stubblefield 2001; Wang and Reiter 2007; Waters et al. 2004]. Others have proposed memory, rather than CPU, cycles for similar purposes [Abadi et al. 2005], and still others use money as the scarce resource [Mankins et al. 2001; Stavrou et al. 2004].

One of the contributions of speak-up is to introduce bandwidth as the scarce resource, or currency. Another contribution is the explicit goal of a resource-proportional allocation. Many of the other proposals strive for a notion of fairness, and a few implicitly achieve a resource-proportional allocation or something close, but none state it as an objective (perhaps because it only makes sense to state this objective after establishing that no robust notion of host identity exists). An exception is a recent paper by Parno et al. [2007], which was published after our work [Walfish et al. 2006].

We do not know of another proposal to use bandwidth as a currency. However, Sherr et al. [2005] and Gunter et al. [2004] describe a related solution to DoS attacks on servers' computational resources. In their solution, 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 [2003] observes that a hypothetical defense based on client retries and timeouts would require less overhead but still provide the same qualitative performance bounds as proof-of-work schemes. Because this general approach does not meet his more exacting performance requirements, he does not consider using bandwidth as a currency.

9.1.1 *Bandwidth vs. CPU.* We chose bandwidth as a currency because we were originally motivated by the key observation at the beginning of Section 4, namely that good clients likely have more spare upload capacity. However, our resource-proportional goal could just as well be applied to CPU cycles, so we must ask, "Why bandwidth? Why not use CPU cycles as the computational currency?" To answer this question, we now compare these two alternatives. We do not find a clear winner. Bandwidth strikes us as the more "natural" choice, but it can introduce more collateral damage.

We begin with a hedge; we discuss contributions of speak-up that are expressed in the context of bandwidth but could apply to other currencies.

Less mechanism, more desiderata. The existing CPU-based proposals [Aura et al. 2000; Juels and Brainard 1999; Feng 2003; Parno et al. 2007; Dean and Stubblefield 2001; Wang and Reiter 2007; Waters et al. 2004] incorporate far more mechanism than speak-up: they require some or all of client modification, protocol modification (to accommodate the puzzle challenge and response), or a network-wide puzzle distribution system together with a trusted authority. We contend that this extra mechanism is unnecessary, at least for protecting Web applications. Consider a hypothetical defense that works just like speak-up, except instead of clients' browsers coughing up dummy bits, their browsers cough up solutions to small, fixed-size CPU puzzles; in each interval, the thinner admits the client that has solved the most number of puzzles. (Similar ideas, with different implementations, are described by Wang and Reiter [2007] and Parno et al. [2007].) Note that this defense would not require the extra mechanisms such as client modification.

Moreover, this hypothetical defense would, like speak-up, have the following desirable properties: it would find the price correctly (by "price," we mean the going rate of access, expressed in CPU cycles or puzzle difficulty level); it would find the price automatically, with no explicit control or adjustment (in some proposals, either the server sends clients puzzles of a particular difficulty level or clients must guess a difficulty level); it would resist gaming; and it would work with unmodified clients. No existing CPU-based proposal has all of these properties. Thus, even if the reader is not convinced by the advantages of bandwidth, speak-up's auction mechanism is useful in CPU-based schemes too.

Advantages of bandwidth. In a head-to-head comparison, bandwidth has two advantages compared to CPU:

1. *Implementation-independent*. A scheme based on bandwidth cannot be gamed by a faster client implementation. In the hypothetical CPU-based defense just given, good clients would solve puzzles in JavaScript; bad ones might use a puzzle-solving engine written in a low-level language, thereby manufacturing an advantage. (Note that our hypothetical CPU-based defense is a conservative benchmark because, as mentioned above, it is an improvement compared to the CPU-based defenses proposed in the literature.)

2. *Bandwidth is attackers' actual constraint*. Today, absent any defense, the apparent limit on DoS-conducting attackers is bandwidth, not CPU power (any

host issuing requests at high rate will saturate its access link long before taxing its CPU). Charging in the resource that is the actual limiting factor yields a benefit: charging any price in that currency (including one that is below what is needed to achieve a resource-proportional allocation) will have some effect on the attackers. For example, assume that bad clients have maxed out their bandwidth and that the "correct" price is 50× as many bits as are in a normal request. In this situation, if the server requires all clients to spend, say, twice as many bits per request rather than 50× as many bits, then it will halve the effective request rate from the bad clients while very probably leaving the good clients unaffected. CPU cycles offer no such benefit: to affect the bad clients at all, the price in CPU cycles must be high. Of course, in our design of speak-up, servers do not state the price as a fixed multiple (doing so would not achieve a resource-proportional allocation), but they could do so in a variant of the scheme.

Disadvantages of bandwidth. Against the advantages above, bandwidth has three disadvantages relative to CPU cycles; we believe that none of them is fatal:

1. *CPU is more local* (*but not completely so*). Bandwidth is shared, in two ways. If a client spends bandwidth, it may adversely affect a client that isn't "speaking up." And, when two "speaking up" clients share a bottleneck link, they also "share" an allocation at the server.

However, these disadvantages may be less pronounced than they seem. First, CPUs are not purely local resources. With the advent of cloud computing and with entire computing sessions happening in virtual machines, a single CPU may be multiplexed over multiple users. Or, two users may be logged into the same computer. Second, with respect to a client's bandwidth consumpion adversely affecting a client that is not speaking up, we observe that an analogous situation holds for any network application that is a heavy bandwidth consumer. For example, BitTorrent is one of the predominant applications on the Internet, is a heavy consumer of clients' upload bandwidth (far more than speak-up, which is only invoked when the server is under attack), and likely causes collateral damage. Yet BitTorrent seems to have gained acceptance anyway.

2. *Asymmetry of CPU puzzles*. Solving a puzzle is slow; checking it is fast. Bandwidth does not have an analogous property: the front-end to the server must sink all of the bandwidth that clients are spending. However, we do not view this disadvantage as fatal; see the discussion of condition C2 in Section 3.

3. *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 could address this disadvantage by changing the implementation of speak-up slightly so that it gives humans the opportunity to express whether they want to pay bandwidth (e.g., one could imagine the thinner exposing the "going rate" in bits and letting customers choose whether to continue).

Stepping back from the comparison between CPU and bandwidth, we wonder whether there is a design that combines the two currencies to get the advantages of both. We leave this question for future work.

9.1.2 *Drawbacks of Resource-Based Schemes.* We now discuss critiques of resource-based schemes in general. First, as discussed in condition C1 in Section 3, any scheme that is trying to achieve a roughly proportional allocation only works if the good clients have enough currency (a point made by Laurie and Clayton in the context of proof-of-work for spam control [2004]).

A second disadvantage that inheres in these schemes is that they are only roughly fair. In the context of speak-up, we call this disadvantage *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 clients' bandwidths, 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 that 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.

A third critique of resource-based approaches in general, and speak-up in particular, is that they treat *flash crowds* (i.e., overload from good clients alone) as no different from an attack. This fact might appear unsettling. However, observe that, for speak-up at least, the critique 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 Section 3). 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.

A final critique of resource-based schemes is that they give attackers *some* service so might be weaker than the schemes that we discuss next 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.

9.2 Detect-and-Block Defenses

The most commonly deployed defense [Network World 2005] is a combination of link over-provisioning (e.g., Prolexic Technologies, Inc. and BT Counterpane) and profiling, which is a detect-and-block approach offered by several vendors (Cisco Guard, Mazu Networks, Inc., Arbor Networks, Inc.). 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. In application-level profiling [Ranjan et al. 2006; Srivatsa et al. 2006], the server gives preference to clients who appear to have "typical" behavior. Resource containers [Banga et al. 1999] perform rate-limiting to allocate the server's resources to clients

fairly (more generally, one can use Fair Queuing [Demers et al. 1995] to rate-limit clients based on their IP addresses). Defenses based on CAPTCHAs [von Ahn et al. 2004] (e.g., [Morein et al. 2003; Google Captcha 2005]) use reverse Turing tests to block bots. Killbots [Kandula et al. 2005] combines CAPTCHAs and rate-limiting, defining a bot as a non-CAPTCHA answering host that sends too many requests to an overloaded server.

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 [Pittsburgh Post-Gazette 2003] to solve the CAPTCHAs) or "good bots" (legitimate, non-human clientele or humans who do not answer CAPTCHAs). As mentioned in Sections 1.1 and 3, schemes that rate-limit clients by IP address can err because of address hijacking and proxies. Profiling apparently addresses some of these shortcomings today (e.g., many legitimate clients behind a proxy would cause the proxy's 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.

9.3 Mechanisms for Blocking Traffic

Much recent research has sought *mechanisms* for blocking traffic destined to servers under attack; the *policies* for such blocking are often unspecified. For this reason, we believe that this research is orthogonal to, and can be combined with, speak-up and the other members of the taxonomy presented at the beginning of the article. We now give more detail.

Examples of proposed mechanisms include the recent literature on capabilities [Yang et al. 2005; Anderson et al. 2003; Yaar et al. 2004]; dFence [Mahimkar et al. 2007], in which server operators can dynamically deploy middleboxes to filter problematic traffic; and an addressing scheme in which hosts that are always clients cannot be addressed [Handley and Greenhalgh 2004] (see the bibliographies of those papers for other examples). These proposals share a similar high-level structure: they describe systems to keep traffic from potential victims—under the assumption that the infrastructure or the protected host knows which packets are worth receiving. For example, with capabilities [Yang et al. 2005; Anderson et al. 2003; Yaar et al. 2004], servers are protected by *capability allocators* that act on their behalf. These allocators give requesting clients tokens, or capabilities, that routers understand. Clients then place the capabilities in the packets that they originate, and routers give such packets higher priority than packets without capabilities.

These techniques focus on how to block traffic, not on which traffic to block. For the latter function, the authors generally suggest detect-and-block techniques (e.g., CAPTCHAs), but they could easily use speak-up, as mentioned in Section 3. For example, when a server is over-subscribed, its capability allocator could conduct a bandwidth auction to decide which clients receive capabilities. Indeed, under the threat in which good and bad are indistinguishable, these proposals would *have* to use speak-up or another resource-based scheme!

As an example, Parno et al. [2007] advocate CPU puzzles for precisely this purpose, but their paper considers bandwidth as a candidate resource.

9.4 Summary

Because detect-and-block defenses can err, we favor resource-based defenses for the threat described in Section 3. We have argued that bandwidth is a natural (but not the only) choice for the resource and that speak-up's mechanisms may still be useful under other resource choices. Finally, we have addressed important critiques of resource-based approaches in general and speak-up in particular.

10. PLAUSIBILITY OF THE THREAT AND CONDITIONS

Though we have argued that speak-up can be an appropriate defense, given the threat and conditions that we modeled in Section 3, we have so far not said to what extent the threat and conditions occur in practice. Now we will address the following questions in turn: (1) How often is the threat manifest? (2) How often does condition C1 apply, that is, how often are aggregate good and bad bandwidths roughly equal? (3) Is condition C2 reasonable, that is, can we assume that servers have adequate link bandwidth? To answer these questions, this section synthesizes the research and anecdotes of others, relying on secondary and tertiary sources.

Of course, if deployed, speak-up would cause attackers to change their tactics, possibly altering the threat. We consider such dynamics, together with the next response from the academic and security communities, at the end of the section (Section 10.4).

10.1 The Threat

By "the threat," we mean requests that are (a) application-level, (b) legitimate-looking, and (c) of uncertain origin. We address these characteristics in turn. For (a), reports are mixed. A dataset from Shadowserver[14] that covers DDoS activity by ∼1500 botnets controlled via Internet Relay Chat (IRC) does not indicate any application-level attacks, but Prolexic Technologies reportedly sees mostly this type [Lyon 2006]. However, in Prolexic's case, the requests are often ill-formed, so characteristic (b) does not hold. For (c), we know that some attackers hijack addresses for sending spam and that proxies are widespread (see Section 3 and Appendix B). Also, bots are ever more sophisticated, and botnets are becoming smaller [Rajab et al. 2007; CNET News 2005; McPherson and Labovitz 2006; Lyon 2006; eWEEK 2006; Cooke et al. 2005; **?**], presumably to fly under the "detection radar" of victims and the mitigation community. The combination of smarter but smaller bots will make the three characteristics above—which require smart bots but which conserve the adversary's resources—more likely.

Regardless, this discussion obscures two larger points. First, even if such attacks have not been observed in their pure form, it is not hard to carry them out: the vulnerability is real. Second, we believe that it is important to be

[14]http://www.shadowserver.org.

proactive, that is, to identify weaknesses before they are exploited. Thus, even if the underground economy does not favor this attack today, we must ask—and try to answer—the question of how to defend against it.

10.2 Relative Sizes of Good and Bad Clientele

We first discuss the sizes of botnets and then discuss the implications for what types of sites speak-up can protect.

We begin by arguing that most botnets today consist of fewer than 100,000 hosts, and even 10,000 hosts is a large botnet. Although there have been reports of botnets of over 100,000 hosts [Handley 2005; Honeynet Project and Research Alliance 2005; Brown 2006; McLaughlin 2004; Dagon et al. 2006] (and millions in the case of a report in TechWeb News [2005]), most botnets are far smaller. Freiling et al., in a study of 180 botnets in 2005, find that the largest botnets were up to 50,000 hosts, with some being just several hundred strong [Freiling et al. 2005, Section 5]. Symantec reports that 2,000 to 10,000 hosts is a common range [McLaughlin 2004]. Shadowserver[15] tracks thousands of botnets and reports a total number of bots in the single millions, implying that the average botnet size is in the low thousands. Cooke et al. [2005] interviewed operators from Tier-1 and Tier-2 ISPs; their subjects indicated that whereas botnets used to have tens of thousands of nodes several years before, sizes of hundreds to thousands of nodes became the norm. Rajab et al. [2006, 2007] find similar numbers—hundreds or thousands—for the "live populations" of various botnets. Indeed, they point out that some studies may report the footprint—the total number of hosts that are infected with the bot—which may account for the larger sizes. For speak-up, we are concerned only with the botnet's *current* firepower and hence with the "live population."

The observed sizes of link flooding attacks back up these rough estimates, as we now argue. First, consider the observations of Sekar et al. [2006] about attacks in a major ISP; the relevant graph from their paper is reproduced in Figure 15, and the relevant line is the solid one. The graph shows that a 100 Mbits/s DoS attack is over the 90th percentile (specifically, it is at the 93rd percentile [Sekar 2007]). And 1 Gbit/s attacks are at the 99.95th percentile [Sekar 2007]. Their study, combined with anecdotes [Sturgeon 2005; Thomas 2005; Handley 2005], suggests that although a small number of link-flooding attacks are over 1 Gbit/s, the vast majority are hundreds of Mbits/s or less. These sizes in turn suggest that the attacks are not being carried out by gigantic botnets. Our reasoning is as follows. One study found that the average bot has roughly 100 Kbits/s of bandwidth [Singh 2006]. Thus, even if each bot uses only a tenth of its bandwidth during an attack, an attack at the 90th (resp., 99.95th) percentile could not have been generated by a botnet of larger than 10,000 (resp., 100,000) nodes. Our conclusion is that most attacks are launched from botnets that are under 10,000 hosts. (Of course, it is possible that the attackers controlled millions of hosts, each of which was sending traffic very slowly, but in that case, *no* defense works.)

[15]http://www.shadowserver.org/wiki/pmwiki.php?n=Stats.BotCounts.

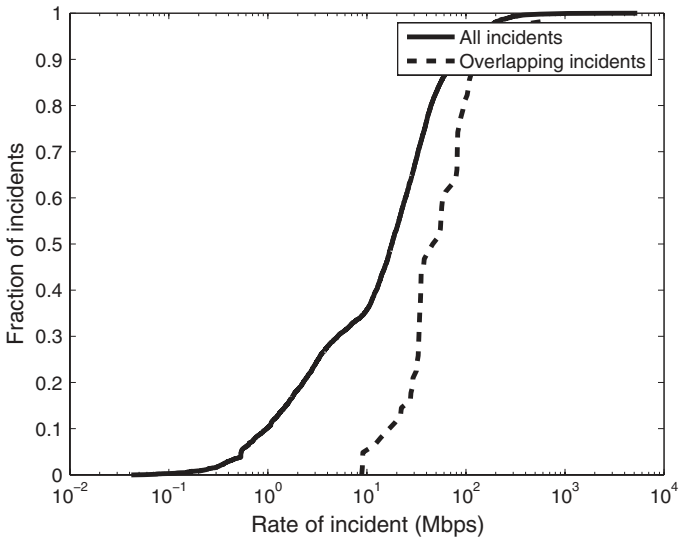

Fig. 15. Rates of potential attack incidents observed by Sekar et al. [2006]. This figure is reproduced, with permission, from Figure 16 of their paper.

Given these numbers, how big does the legitimate clientele have to be to fully withstand attack? As mentioned in Section 2, the answer depends on the server's utilization (or, what is the same thing, its degree of overprovisioning), as we now illustrate. Recall from Section 4.1 that, for the good clients to be unharmed in the ideal case, we must have $c \geq g(1 + B/G)$. Letting $u = g/c$ be the usual utilization of the server, we get $G/B \geq u/(1-u)$. If the bandwidth of a population is proportional to the number of members, then the good clients must have $u/(1-u)$ as many members as the bots to fully withstand attack. Putting this result in context, if a service has spare capacity 90% (i.e., $u = 0.1$), speak-up can fully defend it (i.e., leave its good clients unharmed) against a 1,000-host (resp., 10,000-host) botnet if the good clients number $\sim$100 (resp., $\sim$1,000). If a service has spare capacity 50%, then the good clients must number $\sim$1,000 (resp., $\sim$10,000).

Many sites have clienteles of this order of magnitude: observe that these numbers refer to the good clients currently *interested* in the service, many of which may be quiescent. For example, `http://www.kayak.com`, a computationally intensive travel distribution site, often claims to have at least tens of thousands of clients online. Many of these clients are humans pausing between queries, but, from speak-up's perspective, their machines count in the "current clientele." A more extreme example is Wikipedia, which, according to the Web analytics company Alexa,[16] is one of the top 10 sites on the Web (as of August, 2007). According to Wikipedia's statistics [Weber 2007a], they get an average of 20,000 requests per second. Assuming (likely pessimistically) that a human reading Wikipedia makes a request once every ten seconds, the number of concurrent clients interested in the service is 200,000.

[16]http://www.alexa.com.

10.3 Costs for the Server

We discussed how to satisfy condition C2 in Sections 3 and 5.3. And we showed in Section 8.8 that even if the condition isn't satisfied, speak-up still offers some benefit. Here, we just want to make two points about C2. First, any other scheme that were to seek a proportional allocation would also need condition C1. Thus, C2 represents the relative cost of speak-up to the server. Second, one can regard this cost as paying bandwidth to save application-level resources. We are not claiming that this trade-off is worthwhile for every server, only that speak-up creates such an option and that the option may appeal to some server owners.

10.4 How Will Adversaries Respond to Speak-Up?

We begin by asking, *How are the adversaries likely to respond to speak-up?* If we were in the adversary's position, we would attack speak-up by trying to amass as many resources as possible. More generally, because speak-up achieves a roughly proportional allocation, an attacker's power is given by the number of hosts he controls, so the attacker's response will be to try to acquire more hosts.

And how will the "good guys"—the security community, law enforcement, etc.—respond to those efforts? They will respond by trying to minimize the number of machines available to adversaries. However, their efforts will not be perfect: even if they could eliminate compromised machines altogether, adversaries could still use their own computers to mount attacks.

Given all of this back-and-forth, how can we argue that speak-up will have a positive effect? Recall that speak-up forces attackers to acquire many more machines to conduct the same level of service-denial as they can under the status quo (as stated in Section 2). Thus, the question becomes: how hard is it for an adversary to compromise or control orders of magnitude more machines?

We believe that such compromises would be costly for attackers, if not downright crippling. Our reasons are twofold. First, compromising machines is already a competitive activity; bot herders compete with each other based on how many machines they control. Thus, any given adversary today already has an incentive to compromise as many machines as he can. Second, compromising machines happens automatically (e.g., two ways that worms spread are by scanning and as users open attachments that email themselves to the users' contact lists). Thus, we can assume that for any known exploit, all of the vulnerable machines are already compromised (roughly speaking). The implication is that compromising further machines requires identifying further vulnerabilities. Yet, doing so cannot be easy: if a vulnerability were easy to find, it would have been found already, given the competition mentioned above.

To summarize, while we cannot predict attackers' next steps exactly, we do know that speak-up has increased their costs—and that the increase is likely to be significant.

11. REFLECTIONS

We first summarize speak-up's purpose, its contributions, and its costs, and then discuss it in a broader context—how and when it combines with other defenses, and where else it may apply.

Summary. Our principal finding in this article has been that speak-up mostly meets the goal in Section 4.1—a roughly fair allocation, based on clients' bandwidths. Thus, speak-up upholds the philosophy in Section 1.1.

The contributions of this article are to introduce bandwidth as a computational currency; to articulate the goal of a proportional allocation based on this currency; to give a family of mechanisms to charge in the currency that are simple, that find the correct "price," that do not require clients or the server to know or guess the price, and that resist gaming by adversaries; and to embody the idea in a working system that incorporates one of these mechanisms and works with unmodified Web clients.

Speak-up certainly introduces costs, but they are not as onerous as they might appear. The first set of costs, to the victimized server, we have just discussed (Section 10.3). The second set of costs is to the network, because speak-up introduces traffic when servers are attacked. Yet, every network application introduces traffic (and some, like BitTorrent, introduce a whole lot more traffic). And, as with most other network applications, the traffic introduced by speak-up obeys congestion control (because the main traffic increase from speak-up is from the good clients). The third set of costs is to the end-user: there exist variable bandwidth costs (which we discussed in Section 9.1.1), and, also, speak-up may edge out other activity on the user's upload link. Such an opportunity cost is unfortunate, but opportunity costs apply to all resource-based defenses (e.g., if the server charged in CPU cycles, the user's tasks would compute more slowly).

Thus, while speak-up may not make sense for every site or every denial-of-service attack, we think that it is a reasonable choice for some pairings. The benefit is that the implementation, and the overall idea, are ultimately quite simple. Moreover, the costs that we have been discussing have been for the "worst-case" threat; when that threat is only partially manifest, the costs of speak-up are lower, as we now describe.

Speak-up combined with other defenses. Speak-up is designed for the threat described in Section 3, namely a situation in which the server cannot identify its clients (bad clients mimic good ones). Yet, this threat may not always hold in practice, as mentioned in Section 10, or it may hold only partially. In these cases, one can use other defenses, with speak-up as the fallback or backstop.

Specifically, a server may, in practice, be able to recognize some of its legitimate clientele as such. For example, when a client makes a purchase at a Web server, the server can infer that the client is legitimate and issue a cookie to the client. Then, when the server is overloaded or attacked, it (a) prioritizes clients that present valid cookies but does not charge them bandwidth and (b) runs speak-up for the remainder of its clientele. This approach saves bandwidth,

both for the known clients and for the server (because the known clients are not speaking up). This approach also gives low bandwidth clients a way to overcome their bandwidth disadvantage (discussed in Section 9.1.2)—become known to the server. The disadvantage of this approach is that the unknown legitimate clients will be competing for a smaller piece of the server so may be worse off, compared to a uniform application of speak-up.

Other applications of speak-up. While we have discussed speak-up in the context of defending servers against application-level denial-of-service, it can apply more broadly. We now list two sample applications. First, one could use a variant of speak-up to guard against "Sybil attacks" [Douceur 2002] (i.e., attacks in which clients manufacture identities) in peer-to-peer networks: the protocol could require clients to send large and frequent heartbeat messages; clients would then be unable to "afford" many identities.

Second, in some contexts, speak-up could defend against link attacks: a link is fundamentally a resource and could be protected and allocated just like a server's computational resources. For instance, consider cases in which the "link" is a contained channel reserved for a particular class of traffic (as in the scenario of capability requests, covered by Parno et al. [2007]). One could allocate such a channel roughly fairly—without needing a thinner—by requiring clients to send as much traffic as possible through the channel. Then, the channel would be over-subscribed, and a client's chances of getting bits through the channel would be in proportion to the quantity of traffic that it sends.

APPENDIX

A. QUESTIONS ABOUT SPEAK-UP

A.1 The Threat

How often do application-level attacks occur?
We do not know how often they occur. They seem to be less common than other forms of DDoS but, according to anecdote, the trend is toward such attacks. See Section 10.1 for more detail.

Why would an adversary favor this type of attack?
Such attacks are harder to filter (because existing DDoS-prevention tools already have the ability to filter other kinds of attacks). Moreover, application-level attacks require less bandwidth (because the request rate needed to deplete an application-level resource is often far below what is needed to saturate a server's access link). This latter characteristic would attract adversaries who have access to small botnets or who need to conserve resources, and indeed there are reports that botnets are becoming smaller [Rajab et al. 2007; CNET News 2005; McPherson and Labovitz 2006; Lyon 2006; eWEEK 2006; Cooke et al. 2005; **?**].

If such attacks are not yet common, why bother thinking about the defense?
We answer this question in Sections 1.3 and 10.1. Briefly, there are a few reasons. First, the vulnerability exists and thus could be exploited in the future.

Second, we believe that the trend is toward such attacks. Last, we believe that it is important to be proactive, that is, to identify and defend against weaknesses before they are exploited.

How often are the attacking clients' requests actually indistinguishable from the legitimate clients' requests?
We do not know. According to anecdote, many application-level attacks are primitive and hence easily filtered. However, as argued just above, we believe that the trend is toward smarter attacks and smaller botnets. See Section 10.1.

What makes you think that bad clients send requests at higher rates than legitimate clients do?
If bad clients *weren't* sending at higher rates, then, as long as their numbers didn't dominate the number of good clients, the server could restore service to the good clients with modest over-provisioning. (If the number of bad clients is vastly larger than the number of good clients, and requests from the two populations are indistinguishable, then *no* defense works.)

Aren't there millions of bots? Aren't current DDoS attacks 10 Gbits/s? How can speak-up possibly defend against that?
See Sections 2, 3, and 10.2. In short: first, only a few botnets are of this size and only a minuscule fraction of attacks are 10 Gbits/s (Section 10.2). Second, speak-up (or any resource-based defense) works best when the adversarial and good populations are roughly the same order of magnitude. As mentioned in the answer to the previous question, if the adversarial population vastly outnumbers the good population, and if requests from the two populations are indistinguishable, then *no* defense works.

Can speak-up defend tiny Web sites?
Yes and no. Speak-up helps no matter what. However, speak-up cannot leave the legitimate clientele unharmed by an attack unless the legitimate population and the attacking population are of the same order of magnitude (or unless the server is highly over-provisioned). See Section 2.

A.2 The Costs of Speak-up

Doesn't speak-up harm the network, a communal resource?
See Sections 2, 5.1, and 11. Our brief answer is that speak-up introduces extra traffic only when a server is attacked, that speak-up's traffic is congestion-controlled, that the network core appears to be overprovisioned, and that one should regard speak-up as simply a heavy user of the network. However, as with any application (e.g., BitTorrent), there might be collateral damage from the extra traffic introduced by speak-up.

But bad guys won't control congestion!
True. However, a bad client refusing to control congestion is carrying out a link attack, which speak-up does not defend against (a bad client can carry out such an attack today); see Section 3. And, if a bad client does flood, the bad client won't get much more of the server than it would if it obeyed congestion control (see Section 4.4). Thus, speak-up does its job regardless of whether bad clients control congestion.

What is the effect of speak-up when links are shared?
See Section 5.2.

Bandwidth is expensive, so why would a site want to use speak-up, given that speak-up requires the site to allocate a lot of inbound bandwidth?
The economics of every site are different. For some sites, speak-up is certainly less economical than over-provisioning the server's application-level resources to handle every good and bad request. However, we believe that there are other sites for which bandwidth is not terribly expensive; see condition C2 in Sections 3 and 5.3.

Doesn't speak-up introduce opportunity costs for end-users?
Yes, but such costs are introduced by any network application and, indeed, by any resource-based defense. See Sections 2 and 11.

A.3 The General Philosophy of Speak-Up

Won't speak-up cause adversaries to acquire more resources (i.e., compromised hosts)? For example, if speak-up has nullified a 100-node botnet, won't an adversary just build a 10,000-node botnet? Thus, won't speak-up inflame the bot problem?
It is true that speak-up (or any resource-based defense) creates additional incentive for adversaries to compromise machines. However, the cost to doing so is likely quite high: we believe that many of the computers worldwide that could be compromised cheaply already have been. Thus, speak-up increases the adversary's costs, thereby resulting in a higher fence. We discuss this point in Sections 2 and 10.4.

Doesn't speak-up give ISPs an incentive to encourage botnets as a way to increase the bandwidth demanded by good clients?
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.

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.

A.4 Alternate Defenses

Instead of charging clients bandwidth and allocating the server in a way that is roughly fair, why not allocate the server in a way that is explicitly fair by giving every client the same piece?
Doing so requires that the server be able to identify its clients. However, our threat model presumes that, given a request, the server cannot be sure which client originated it. The reasons for such uncertainty are address hijacking, proxies, and NAT. For more detail, see Section 1 and Section 3.

Why does speak-up make clients consume *bandwidth? Why doesn't speak-up simply determine how much bandwidth each of its clients has (say, by using a bandwidth-probing tool) and then allocate the server according to this determination?*
Doing so would again require that the server be able to identify its clients. For example, if an adversary adopts several IP addresses, each of these addresses would appear to have the same bandwidth, thereby giving the adversary a bandwidth advantage.

Instead of charging clients bandwidth, why not charge them CPU cycles?
Such a defense would be a reasonable alternative. For a detailed comparison of bandwidth and CPU cycles as computational currencies, see Section 9.1.1.

Sites need to protect themselves against link attacks (and speak-up does not serve this purpose, as you state in Section 3). So why not regard the application as being connected to a virtual link, and use existing link defenses to protect the application?
Depending on the scenario, this approach may work. However, deploying link defenses often requires network modification or help from ISPs; it may be easier to deal with application-level attacks on one's own. Also, many defenses against link attacks work by detecting very high amounts of aggregate traffic, and an effective application-level attack needs far less bandwidth so may not trigger these defenses. Finally, adversaries may move away from link attacks, removing the need for link defenses.

How does speak-up compare to ... ?
Please see Section 9. In that section, we compare speak-up to many other defenses.

A.5 Details of the Mechanism

Can bandwidth be faked? For example, can a client compress its bandwidth payment to give the illusion that it is paying more bits? Or, could a client get a proxy to pay bits on its behalf?
The thinner counts the bits that arrive on behalf of a request, so a client connecting directly to the thinner cannot fake its bandwidth payment. And, proxies generally relay what clients send, so if a client compresses its payment en route to the proxy, then the proxy will submit a compressed request to the thinner.

By how much does speak-up increase bandwidth consumption?
Assume that the bad clients were flooding independent of speak-up. In the worst case, the good clients need to spend all of their bandwidth. In this case, the extra bandwidth consumption is $G/(gn)$, where G is the total bandwidth of the good clients expressed in bits/s, g is the good clients' legitimate demand expressed in requests/s, and n is the size of a request, in bits. However, if the good clients do not need to spend all of their bandwidth, the bandwidth consumption may be far smaller; see the discussion of "price" in Section 4.2.

Does speak-up allow bad clients to amplify their impact? For example, if bad clients attack a site, they can trigger speak-up, causing the good *clients to pay*

bandwidth. Don't adversaries therefore have a disproportionate ability to increase traffic under speak-up?
At a high level, the answer is no. The "price"—that is, the extra traffic introduced by speak-up—varies with the attack size. Roughly, if the bad clients do not spend much bandwidth, then they do not make the good clients spend much bandwidth. Like the previous question, this one is related to the discussion of "price" in Section 4.2.

Can the bad clients amplify their impact by cycling through a population of servers, driving each into overload but spending only a little bit of time at each server?
No. The purpose of speak-up is exactly to give clients service in proportion to the bandwidth that they spend. Thus, if the bad clients go from site to site, never spending much bandwidth at any site, then they will not get much service. Moreover, if they follow this pattern, then the price at each of the sites will be low, so the good clients will not spend much bandwidth either. If the bad clients do spend many bits and temporarily drive up the price at a site, that price will subside once those bad clients leave.

A.6 Attacks on the Thinner

What happens if the thinner gets a lot of clients or connections at once? Can it run out of file descriptors?
Yes, but the implementation protects itself against this possibility. The implementation recycles file descriptors that correspond to requests that have not been "active" for some period. Moreover, we have configured the thinner to allocate up to hundreds of thousands of file descriptors; see Section 7.

Is it possible to build a production-quality thinner, given that the box would have to sink bits at high rate and maintain a lot of state?
Yes, we believe it is possible. The state for each request is very little—a TCP control block, a counter for the number of bytes that have been paid on behalf of that request, and a small amount of other per-request data. Moreover, even our unoptimized implementation can sink bits at a high rate; see Section 8.1.

A.7 Other Questions

What happens if a server defended by speak-up experiences a flash crowd, that is, overload from legitimate clients?
Such a server will conduct a bandwidth auction, just as if it were under attack. Though this fact might seem distasteful, observe that if the server is overloaded it still has to decide which requests to drop. Deciding based on bandwidth is not necessarily worse than making random choices. We discuss this question in more detail when critiquing resource-based schemes (see Section 9.1.2).

Does your implementation of speak-up work if the Web client does not run JavaScript?
No. See Section 7.

Under speak-up, all clients are encouraged to send at high rates. So how is one supposed to tell the difference between the legitimate clients and the bots?

An attacked Web site cannot tell; indeed, part of the motivation for speak-up is that it can be difficult for Web sites to identify clients. However, the *ISP* of a given bot should be able to tell, based on the computer's traffic consumption: for a good client, most of the sites that it visits are not under attack, whereas a bot consumes much more of its access link. Thus, if it were economically attractive for ISPs to identify and eliminate bots, they could do so.

B. ADDRESS HIJACKING

In this appendix, we describe how an adversary can temporarily adopt IP addresses that it does not own. The attack can take various forms.

Spurious BGP Advertisements. An attacker can issue spurious Border Gateway Protocol (BGP) advertisements, thereby declaring to the Internet's routing infrastructure that the attacker owns a block of IP addresses to which he is not entitled. The result is that the attacker may now be reachable at potentially millions of IP addresses. This attack works because many ISPs (a) accept BGP routes without validating their provenance and (b) propagate these routes to other ISPs [Feamster et al. 2005]. Such spurious adoption of IP addresses has been observed before [Boothe et al. 2006] and correlated with spam transmissions [Ramachandran and Feamster 2006].

Stealing from the Local Subnet. A host can steal IP addresses from its subnet [Falk 2006]; this attack is particularly useful if the thieving host is a bot in a sparsely populated subnet. The attack works as follows:

—The thieving host, H, cycles through the IP addresses in the subnet. For each IP address X, H broadcasts an ARP (Address Resolution Protocol) request for X.
—If H does not receive a reply to this ARP, H infers (assumes) that X is currently unused. At this point, H undertakes to steal X, using one of two methods:
  —H sends packets to remote destinations, and the packets have source IP address X. Any reply packet will be addressed to X. When these reply packets arrive at the subnet, the router in front of the subnet will issue an ARP request for X. H simply responds to this request, declaring to the subnet that it is reachable at IP address X.
  —Another option is for H to preempt the process above by broadcasting an ARP *reply* associating X with H's hardware address. The local router now believes that H is reachable at IP address X. At this point, H can place source IP address X in its outbound packets and receive replies that are sent to X.

Of course, if the router in front of the subnet is properly configured, this attack is precluded, but not all routers are properly configured.

Stealing from Remote Networks. A host can adopt IP addresses from another network, assuming that the adversary controls hosts in two networks. The goal of this attack is to have a high-bandwidth host (say, one that is owned by the

attacker) appear to have many IP addresses (say, that were harvested from bots on low-bandwidth dial-up connections). The attack works as follows:

—The high-bandwidth, thieving host, H, sends packets with source IP address X; this time, X is the IP address of, for example, a bot with low bandwidth.
—Replies to this packet will go to the bot (these replies, in the case of TCP traffic, will usually be ACKs of small packet size).
—To complete the attack, the bot sends the ACKs to the high-bandwidth host, at which point it has become reachable at IP address X.

In fact, H could combine this attack with the previous one, as follows. The bot could steal X from its subnet, using the previous attack. Then, H and the bot would execute the attack just described.

ACKNOWLEDGMENTS

We thank Vyas Sekar and his coauthors [Sekar et al. 2006] for permission to reproduce Figure 15. For useful comments, critiques, and conversations, we thank Ben Adida, Dave Andersen, Trevor Blackwell, Micah Brodsky, Russ Cox, Jon Crowcroft, Jeff Erickson, Nick Feamster, Frans Kaashoek, Sachin Katti, Eddie Kohler, Christian Kreibich, Maxwell Krohn, Karthik Lakshminarayanan, Vern Paxson, Adrian Perrig, Max Poletto, Srini Seshan, Sara Su, Arvind Thiagarajan, Keith Winstein, Andrew Warfield, the SIGCOMM anonymous reviewers, and Emulab.

REFERENCES

ABADI, M., BURROWS, M., MANASSE, M., AND WOBBER, T. 2005. Moderately hard, memory-bound functions. *ACM Trans. Inter. Tech. 5*, 2.

AGARWAL, S., DAWSON, T., AND TRYFONAS, C. 2003. DDoS mitigation via regional cleaning centers. Sprint ATL Res. rep. RR04-ATL-013177.

ANDERSON, T., ROSCOE, T., AND WETHERALL, D. 2003. Preventing Internet denial-of-service with capabilities. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*.

AURA, T., NIKANDER, P., AND LEIWO, J. 2000. DoS-resistant authentication with client puzzles. In *Proceedings of the International Workshop on Security Protocols*.

BACK, A. 2002. Hashcash—a denial of service counter-measure. `http://www.cypherspace.org/adam/hashcash/hashcash.pdf`.

BALAKRISHNAN, H., RAHUL, H. S., AND SESHAN, S. 1999. An integrated congestion management architecture for Internet hosts. In *Proceedings of the ACM SIGCOMM Conference*.

BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. 1999. Resource containers: A new facility for resource management in server systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

BOOTHE, P., HIEBERT, J., AND BUSH, R. 2006. Short-lived prefix hijacking on the Internet. Presentation to NANOG. `http://www.nanog.org/mtg-0602/pdf/boothe.pdf`.

BROWN, D. 2006. Gangsters hijack home PCs to choke internet with spam. *The Times*. `http://business.timesonline.co.uk/tol/business/law/public_law/article649541.ece`.

CNET NEWS. 2005. Bots slim down to get tough. `http://news.com.com/Bots+slim+down+to+get+tough/2100-7355_3-5956143.html`.

COOKE, E., JAHANIAN, F., AND MCPHERSON, D. 2005. The zombie roundup: Understanding, detecting and disrupting botnets. In *Proceedings of the USENIX Conference on Steps to Reducing Unwanted Traffic on the Internet Workshop (SRUTI)*.

cyberslam. 2004. Criminal complaint filed Aug. 25, 2004, United States v. Ashley et al., No. 04 MJ 02112 (Central District of California). `http://www.reverse.net/operationcyberslam.pdf`.

DAGON, D., ZOU, C., AND LEE, W. 2006. Modeling botnet propagation using time zones. In *Proceedings of the Conference on Network and Distributed System Security Symposium (NDSS)*.

DEAN, D. AND STUBBLEFIELD, A. 2001. Using client puzzles to protect TLS. In *Proceedings of the USENIX Security Symposium*.

DEMERS, A., KESHAV, S., AND SHENKER, S. 1995. Analysis and simulation of a fair queuing algorithm. *ACM SIGCOMM Comput. Comm. Rev. 25*, 1.

DOUCEUR, J. 2002. The sybil attack. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*.

DWORK, C., GOLDBERG, A., AND NAOR, M. 2003. On memory-bound functions for fighting spam. In *Proceedings of CRYPTO*.

DWORK, C. AND NAOR, M. 1992. Pricing via processing or combatting junk mail. In *Proceedings of CRYPTO*.

EWEEK. 2006. Money bots: Hackers cash in on hijacked PCs. `http://www.eweek.com/article2/0,1895,2013957,00.asp`.

FALK, E. 2006. New host cloaking technique used by spammers. `http://thespamdiaries.blogspot.com/2006/02/new-host-cloaking-technique-used-by.html`.

FEAMSTER, N., JUNG, J., AND BALAKRISHNAN, H. 2005. An empirical study of "bogon" route advertisements. *ACM SIGCOMM Comput. Comm. Rev. 35*, 1.

FENG, W. 2003. The case for TCP/IP puzzles. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*.

FRALEIGH, C., MOON, S., LYLES, B., COTTON, C., KHAN, M., MOLL, D., ROCKELL, R., SEELY, T., AND DIOT, C. 2003. Packet-level traffic measurements from the Sprint IP backbone. *IEEE Netw. 17*, 6.

FREILING, F. C., HOLZ, T., AND WICHERSKI, G. 2005. Botnet tracking: Exploring a root-cause methodology to prevent distributed denial-of-service attacks. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*.

GLIGOR, V. D. 2003. Guaranteeing access in spite of distributed service-flooding attacks. In *Proceedings of the International Workshop on Security Protocols*.

Google Captcha. 2005. Stupid Google virus/spyware CAPTCHA page. `http://plo.hostingprod.com/@spyblog.org.uk/blog/2005/06/13/stupid-google-virusspyware-cap.html`.

GUNTER, C. A., KHANNA, S., TAN, K., AND VENKATESTH, S. 2004. DoS protection for reliably authenticated broadcast. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

HANDLEY, M. 2005. In a presentation to Internet architecture working group, DoS-resistant Internet subgroup.

HANDLEY, M. AND GREENHALGH, A. 2004. Steps towards a DoS-resistant Internet architecture. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*.

HONEYNET PROJECT AND RESEARCH ALLIANCE. 2005. Know your enemy: Tracking botnets. `http://www.honeynet.org/papers/bots/`.

JUELS, A. AND BRAINARD, J. 1999. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the Conference on Network and Distributed System Security Symposium (NDSS)*.

KANDULA, S., KATABI, D., JACOB, M., AND BERGER, A. 2005. Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

KNUTH, D. E. 1998. *The Art of Computer Programming* 3rd Ed., Vol. 2. Addison-Wesley, Chapter 3.4.2.

KOHLER, E., HANDLEY, M., AND FLOYD, S. 2006. Designing DCCP: Congestion control without reliability. In *Proceedings of the ACM SIGCOMM Conference*.

KROHN, M. 2004. Building secure high-performance Web services with OKWS. In *Proceedings of the USENIX Technical Conference*.

LAURIE, B. AND CLAYTON, R. 2004. "Proof-of-Work" proves not to work; version 0.2. `http://www.cl.cam.ac.uk/users/rnc1/proofwork2.pdf`.

LYON, B. 2006. Private communication.

MAHIMKAR, A., DANGE, J., SHMATIKOV, V., VIN, H., AND ZHANG, Y. 2007. dFence: Transparent network-based denial of service mitigation. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

MANKINS, D., KRISHNAN, R., BOYD, C., ZAO, J., AND FRENTZ, M. 2001. Mitigating distributed denial of service attacks with dynamic resource pricing. In *Proceedings of the IEEE Computer Security Applications Conference*.

MAZIÈRES, D. 2001. A toolkit for user-level file systems. In *Proceedings of the USENIX Technical Conference*.

MCLAUGHLIN, L. 2004. Bot software spreads, causes new worries. *IEEE Distrib. Syst. Online 5*, 6. http://csdl2.computer.org/comp/mags/ds/2004/06/o6001.pdf.

MCPHERSON, D. AND LABOVITZ, C. 2006. Worldwide infrastructure security report, vol. II. Arbor Networks, Inc. http://www.arbor.net/downloads/worldwide_infrastructure_security_report_sept06.pdf.

MIRKOVIC, J. AND REIHER, P. 2004. A taxonomy of DDoS attacks and DDoS defense mechanisms. *ACM SIGCOMM Comput. Comm. Rev. 34*, 2.

MOREIN, W., STAVROU, A., COOK, D., KEROMYTIS, A., MISHRA, V., AND RUBENSTEIN, D. 2003. Using graphic Turing tests to counter automated DDoS attacks against Web servers. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

NETWORK WORLD. 2005. Extortion via DDoS on the rise. http://www.networkworld.com/news/2005/051605-ddos-extortion.html.

PARK, K., PAI, V. S., LEE, K.-W., AND CALO, S. 2006. Securing Web service by automatic robot detection. In *Proceedings of the USENIX Technical Conference*.

PARNO, B., WENDLANDT, D., SHI, E., PERRIG, A., MAGGS, B., AND HU, Y.-C. 2007. Portcullis: Protecting connection setup from denial-of-capability attacks. In *Proceedings of the ACM SIGCOMM Conference*.

PITTSBURGH POST-GAZETTE. 2003. CMU student taps brain's game skills. http://www.post-gazette.com/pg/03278/228349.stm.

RAJAB, M. A., ZARFOSS, J., MONROSE, F., AND TERZIS, A. 2006. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the ACM Internet Measurement Conference (IMC)*.

RAJAB, M. A., ZARFOSS, J., MONROSE, F., AND TERZIS, A. 2007. My botnet is bigger than yours (maybe, better than yours): why size estimates remain challenging. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Understanding Botnets (HotBots)*. http://www.usenix.org/events/hotbots07/tech/full_papers/rajab/rajab.pdf.

RAMACHANDRAN, A. AND FEAMSTER, N. 2006. Understanding the network-level behavior of spammers. In *Proceedings of the ACM SIGCOMM Conference*.

RAMASUBRAMANIAN, V. AND SIRER, E. G. 2004. The design and implementation of a next generation name service for the Internet. In *Proceedings of the ACM SIGCOMM Conference*.

RANJAN, S., SWAMINATHAN, R., UYSAL, M., AND KNIGHTLY, E. W. 2006. DDoS-resilient scheduling to counter application layer attacks under imperfect detection. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Society*.

RATLIFF, E. 2005. The zombie hunters. *The New Yorker*.

REGISTER. 2003. East European gangs in online protection racket. http://www.theregister.co.uk/2003/11/12/east_european_gangs_in_online.

RHEA, S., GODFREY, B., KARP, B., KUBIATOWICZ, J., RATNASAMY, S., SHENKER, S., STOICA, I., AND YU, H. 2005. OpenDHT: A public DHT service and its uses. In *Proceedings of the ACM SIGCOMM Conference*.

SECURITYFOCUS. 2004. FBI busts alleged DDoS mafia. http://www.securityfocus.com/news/9411.

SEKAR, V. 2007. Private communication.

SEKAR, V., DUFFIELD, N., SPATSCHECK, O., VAN DER MERWE, J., AND ZHANG, H. 2006. LADS: Large-scale automated DDoS detection system. In *Proceedings of the USENIX Technical Conference*.

SHERR, M., GREENWALD, M., GUNTER, C. A., KHANNA, S., AND VENKATESH, S. S. 2005. Mitigating DoS attack through selective bin verification. In *Proceedings of the 1st Workshop on Secure Network Protocols*.

SINGH, K. K. 2006. Botnets—An introduction. Course Project, CS6262, Georgia Institute of Technology. http://www-static.cc.gatech.edu/classes/AY2006/cs6262_spring/botnets.ppt.

SRIVATSA, M., IYENGAR, A., YIN, J., AND LIU, L. 2006. A middleware system for protecting against application level denial of service attacks. In *Proceedings of the ACM/IFIP/USENIX International Middleware Conference*.

STAVROU, A., IOANNIDIS, J., KEROMYTIS, A. D., MISRA, V., AND RUBENSTEIN, D. 2004. A pay-per-use DoS protection mechanism for the Web. In *Proceedings of the International Conference on Applied Cryptography and Network Security*.

STURGEON, W. 2005. Denial of service attack victim speaks out. http://management.silicon.com/smedirector/0,39024679,39130810,00.htm.

TECHWEB NEWS. 2005. Dutch botnet bigger than expected. http://informationweek.com/story/showArticle.jhtml?articleID=172303265.

THOMAS, D. 2005. Deterrence must be the key to avoiding DDoS attacks. http://www.vnunet.com/computing/analysis/2137395/deterrence-key-avoiding-ddos-attacks.

VASUDEVAN, R., MAO, Z. M., SPATSCHECK, O., AND VAN DER MERWE, J. 2006. Reval: A tool for real-time evaluation of DDoS mitigation strategies. In *Proceedings of the USENIX Technical Conference*.

VITTER, J. S. 1985. Random sampling with a reservoir. *ACM Trans. Math. Softw. 11*, 1.

VON AHN, L., BLUM, M., AND LANGFORD, J. 2004. Telling humans and computers apart automatically. *Comm. ACM 47*, 2.

WALFISH, M., BALAKRISHNAN, H., KARGER, D., AND SHENKER, S. 2005. DoS: Fighting fire with fire. In *Proceedings of the ACM Workshop on Hot Topics in Networks (HotNets)*.

WALFISH, M., VUTUKURU, M., BALAKRISHNAN, H., KARGER, D., AND SHENKER, S. 2006. DDoS defense by offense. In *Proceedings of the ACM SIGCOMM Conference*.

WANG, X. AND REITER, M. K. 2007. A multi-layer framework for puzzle-based denial-of-service defense. *Int. J. Inform. Secur.* Forthcoming and published online http://dx.doi.org/10.1007/s10207-007-0042-x.

WATERS, B., JUELS, A., HALDERMAN, J. A., AND FELTEN, E. W. 2004. New client puzzle outsourcing techniques for DoS resistance. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

WEBER, L. 2007a. Wikimedia request statistics. http://tools.wikimedia.de/~leon/stats/reqstats.

WEBER, L. 2007b. Wikimedia traffic statistics. http://tools.wikimedia.de/~leon/stats/trafstats.

YAAR, A., PERRIG, A., AND SONG, D. 2004. SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks. In *Proceedings of the IEEE Symposium on Security and Privacy*.

YANG, X., WETHERALL, D., AND ANDERSON, T. 2005. A DoS-limiting network architecture. In *Proceedings of the ACM SIGCOMM Conference*.

Received February 2008; revised August 2009; accepted January 2010