A DoS-limiting Network Architecture

Xiaowei Yang
University of California, Irvine
xwy@ics.uci.edu

David Wetherall
University of Washington
djw@cs.washington.edu

Thomas Anderson
University of Washington
tom@cs.washington.edu

ABSTRACT

We present the design and evaluation of TVA, a network architecture that limits the impact of Denial of Service (DoS) floods from the outset. Our work builds on earlier work on capabilities in which senders obtain short-term authorizations from receivers that they stamp on their packets. We address the full range of possible attacks against communication between pairs of hosts, including spoofed packet floods, network and host bottlenecks, and router state exhaustion. We use simulation to show that attack traffic can only degrade legitimate traffic to a limited extent, significantly outperforming previously proposed DoS solutions. We use a modified Linux kernel implementation to argue that our design can run on gigabit links using only inexpensive off-the-shelf hardware. Our design is also suitable for transition into practice, providing incremental benefit for incremental deployment.

Categories and Subject Descriptors

C.2.6 [**Computer-Communication Networks**]: Internetworking

General Terms

Security, Design

Keywords

Denial-of-Service, Internet

1. INTRODUCTION

Denial of Service (DoS) attacks have become an increasing threat to the reliability of the Internet. An early study showed that DoS attacks occurred at a rate of nearly 4000 attacks per week [18]. In 2001, a DoS attack [4] was able to take down seven of the thirteen DNS root servers. And more recently, DoS attacks have been used for online extortion [5].

The importance of the problem has led to a raft of proposed solutions. Researchers have advocated filtering to prevent the use of spoofed source addresses [8], traceback to locate the source of the disrupting packets [20, 22, 21, 24], overlay-based filtering [12, 1, 14] to protect approaches to servers, pushback of traffic filters into the network [16, 10, 3], address isolation to distinguish client and server traffic [9], and capabilities to control incoming bandwidth [2, 25].

SIGCOMM'05, August 22–26, 2005, Philadelphia, Pennsylvania, USA.

Each of these proposals has merit and provides techniques that can help address the DoS problem. Yet we argue that each addresses only an aspect of DoS rather than the overall problem. In contrast, our goal is to provide a comprehensive solution to the DoS problem. We require that a DoS-limiting network architecture ensure that any two legitimate nodes be able to effectively communicate despite the arbitrary behavior of k attacking hosts. We limit ourselves to open networks, such as the Internet, where the communicating hosts are not known in advance; this rules out statically configured networks that, for example, only permit predetermined legitimate hosts to send packets to each other.

Our solution is the Traffic Validation Architecture (TVA[1]). TVA is based on the notion of capabilities that we advocated in earlier work [2] and which were subsequently refined by Yaar *et. al.* [25]. Our attraction to capabilities is that they cut to the heart of the DoS problem by allowing destinations to control the packets they receive. However, capabilities are currently little understood at a detailed level or leave many important questions unanswered. A key contribution of our work is the careful design and evaluation of a more complete capability-based network architecture. TVA counters a broader set of possible attacks, including those that flood the setup channel, that exhaust router state, that consume network bandwidth, and so forth.

We have also designed TVA to be practical in three key respects. First, we bound both the computation and state needed to process capabilities. We report on an implementation that suggests our design will be able to operate at gigabit speeds with commodity hardware. Second, we have designed our system to be incrementally deployable in the current Internet. This can be done by placing inline packet processing boxes at trust boundaries and points of congestion, and upgrading collections of hosts to take advantage of them. No changes to Internet routing or legacy routers are needed, and no cross-provider relationships are required. Third, our design provides a spectrum of solutions that can be mixed and matched to some extent. Our intent is to see how far it is possible to go towards limiting DoS with a practical implementation, but we are pragmatic enough to realize that others may apply a different cost-benefit tradeoff.

The remainder of this paper discusses our work in more detail. We discuss related work in Section 2, motivating our approach in Section 3. Section 4 describes a concrete implementation of our architecture and illustrates its behavior. Sections 5, 6 and 7 evaluate our approach using a combination of simulation, a kernel implementation, and analysis. Section 8 discusses deployment issues. Section 9 summarizes our work.

[1]The name TVA is inspired by the Tennessee Valley Authority, which operates a large-scale network of dams to control flood damage, saving more than $200 million annually.

2. BACKGROUND

Our design leverages previous proposals to address DoS attacks. We discuss this work, using attack scenarios to illustrate its strengths and weaknesses and to motivate our approach.

Early work in the area of DoS sought to make all sources identifiable, e.g., ingress filtering [8] discards packets with widely spoofed addresses at the edge of the network, and traceback uses routers to create state so that receivers can reconstruct the path of unwanted traffic [20, 21, 22]. This is a key step, but it is insufficient as a complete solution. For example, an attacker might flood a link between a source and a destination. She might then hide her tracks by using the IP TTL field to cause the attack packets to be dropped after they have done their damage but before they arrive at the destination. The network then simply appears to be broken from the point of view of the source and the destination.

One might think that fair queuing [6] would help by ensuring that each flow gets its fair share of the bottleneck bandwidth. Yet even if we assume ingress filtering (so there are no spoofed source addresses) k hosts attacking a destination limit a good connection to $1/k$ of the bandwidth. This applies even if the destination knows it does not want any of the attack traffic. The problem is worse if fair queuing is performed across source and destination address pairs. Then, an attacker in control of k well-positioned hosts can create a large number of flows to limit the useful traffic to only $1/k^2$ of the congested link. For example, 30 well-placed hosts could cut a gigabit link to only a megabit or so of usable traffic.

A different tack is for the network to limit communication to previously established patterns, e.g., by giving legitimate hosts an authenticator off-line that permits them to send to specific destinations. SOS [12] and Mayday [1] take this approach. Our goal is to design an open network, one where any two hosts can communicate without prior arrangement. Our work can thus be seen as automating the provision of authenticators.

An insidious aspect of the Internet model is that receivers have no control over the resources consumed on their behalf: a host can receive (and have to pay for!) a repetitive stream of packets regardless of whether they are desired. One response is to install packet filters at routers upstream from the destination to cause unwanted packets to be dropped in the network before they consume the resources of the destination, e.g., pushback [16, 10] and more recently AITF [3]. Unfortunately, these filters will block some legitimate traffic from the receiver because there is no clean way to discriminate attack traffic from other traffic, given that attackers can manufacture packets with contents of their choosing. Our work can be seen as a robust implementation of network filtering.

In earlier work, we proposed the approach of putting a capability, or token, into each data packet to demonstrate that the packet was requested by the receiver [2]. Communication takes two steps: first, the sender requests permission to send; after verifying the sender is good, the receiver provides it with a token. When included in a packet, this token allows the network to verify that the packet was authorized by the receiver. By itself, this does not prevent attacks against the initial request packet, the router state or computation needed to verify the packet, and so forth. For example, in our initial work we used a separate overlay for transmitting the request packets; an attack against this channel would disrupt hosts that had not yet established a capability to send.

In SIFF, Yaar *et. al.* refine the capability approach to eliminate the separate overlay channel for request packets and per-flow state. Instead, routers stamp packets with a key that reaches the receiver and is returned to authorize the sender, which uses it on subsequent packets [25]. This is reminiscent of work in robust admission control [15]. We adopt this approach, with some enhancements motivated by weaknesses of the SIFF proposal. First, in SIFF, router stamps are embedded in normal IP packets, which requires each router stamp to be extremely short (2 bits), and thus potentially discoverable by brute-force attack. We show how to combine the security of long stamps with the efficiency of short stamps. Second, initial request packets are forwarded with low priority. This allows attacking hosts to establish "approved" connections purely amongst themselves and flood a path and prevent any further connections from being established along its congested links. We address this through a more careful treatment of request packets. Finally, routers allow all copies of packets with a valid stamp through because they have no per-flow state. Thus, an attacker that is incorrectly granted a capability by a receiver can flood the receiver at an arbitrary rate until the permission expires. This is problematic because a typical Web server will only know after a connection starts whether the traffic is legitimate and given the timeout constants suggested in [25], even a small rate of incorrect decisions would allow DoS attacks to succeed. Our approach is to provide fine-grained control over how many packets can be sent based on a single authorization.

In summary, existing proposed DoS solutions have a number of good ideas but are incomplete. Our goal is to provide a more comprehensive solution. We further restrict ourselves to solutions that might be practically implemented in today's technology, e.g., with limited state at each router and with reasonable amount of computation at each hop.

3. TVA DESIGN OVERVIEW

In this section, we motivate the key components of our design. Later in Section 4, we describe the protocol and sketch its common case of operation. The overall goal of TVA is to strictly limit the impact of packet floods so that two hosts can communicate despite attacks by other hosts. To achieve this, we start with standard IP forwarding and routing. We then extend hosts and routers with the handling described below, conceptually at the IP level. For simplicity of exposition, we consider a network in which all routers and hosts run our protocol. However, our design only requires upgrades at network locations that are trust boundaries or that experience congestion.

3.1 Packets with Capabilities

To prevent a destination from losing connectivity because of a flood of unwanted packets, the network must discard those packets before they reach a congested link. Otherwise the damage has already been done. This in turn requires that routers have a means of identifying wanted packets and providing them with preferential service. To cleanly accomplish this, we require that each packet carry information that each router can check to determine whether the packet is wanted by the destination. We refer to this explicit information as a capability [2].

Capabilities have significant potential benefits compared to other schemes that describe unwanted packets using implicit features [16, 10]. They do not require a difficult inference problem to be solved, are precise since attackers cannot spoof them, and are not foiled by end-to-end encryption. However, to be viable as a solution, capabilities must meet several implied requirements. First, they must be granted by the destination to the sender, so that they can be stamped on packets. This raises an obvious bootstrap issue, which we address shortly. Second, capabilities must be unforgeable and not readily transferable across senders or destinations. This is to prevent attackers from stealing or sharing valid capabilities. Third, routers must be able to verify capabilities without trusting hosts. This ensures malicious hosts cannot spoof capabilities. Fourth, ca-

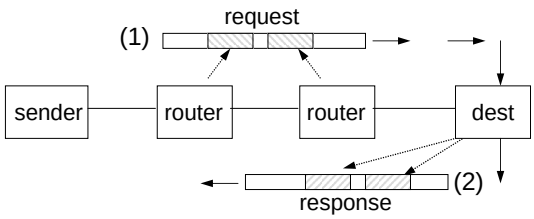

Figure 1: A sender obtaining initial capabilities by (1) sending a request to the destination, to which routers add pre-capabilities; and (2) receiving a response, to which the destination added capabilities.

pabilities must expire so that a destination can cut off a sender from whom it no longer wants to receive packets. Finally, to be practical, capabilities must add little overhead in the common case. The rest of our design is geared towards meeting these requirements.

3.2 Bootstrapping Capabilities

In our design, capabilities are initially obtained using request packets that do not have capabilities. These requests are sent from a sender to a destination, e.g., as part of a TCP SYN packet. The destination then returns capabilities to the sender if it chooses to authorize the sender for further packets, e.g., piggybacked on the TCP SYN/ACK response. This is shown in Figure 1 for a single direction of transfer; each direction is handled independently, though requests and responses in different directions can be combined in one packet. Once the sender has capabilities, the communications is bootstrapped in the sense that the sender can send further packets with capabilities that routers can validate.

Ignoring legacy issues for the moment, we expect the number of packets without associated capabilities to be small in most settings. This is because one capability covers all connections between two hosts, and new capabilities for a long transfer can be obtained using the current capability before it expires. Nonetheless, it is crucial that the initial request channel not open an avenue for DoS attacks, either by flooding a destination or blocking the requests of legitimate senders. The first issue is straightforward to address: we rate-limit requests at all network locations so that they cannot consume all of the bandwidth. Request packets should comprise only a small fraction of bandwidth. Even with 250 bytes of request for a 10KB flow, request traffic is 2.5% of the bandwidth. This allows us to rate-limit request traffic to be no more than 5% of the capacity of each link, with the added margin for bursts.

It is more challenging to prevent requests from attackers from overwhelming requests from legitimate clients. Ideally, we would like to use per-source fair queuing to ensure that no source can overwhelm others, regardless of how many different destinations it contacts. However, this is problematic because source addresses may be spoofed, but per-source fair queuing requires an authenticated source identifier. One possibility is ingress filtering, but we discarded it as too fragile because a single unprotected ingress allows remote spoofing. Another possibility is to sign packets using a public key infrastructure, but we discarded it as too much of a deployment hurdle.

Instead, we build a path identifier analogous to Pi [24] and use it as an approximate source locator. Each router at the ingress of a trust boundary, e.g., AS edge, tags the request with a small (16 bit) value derived from its incoming interface that is likely to be unique across the trust boundary, e.g., a pseudo-random hash. This tag identifies the upstream party. Routers not at trust boundaries do not tag requests as the upstream has already tagged. The tags act as

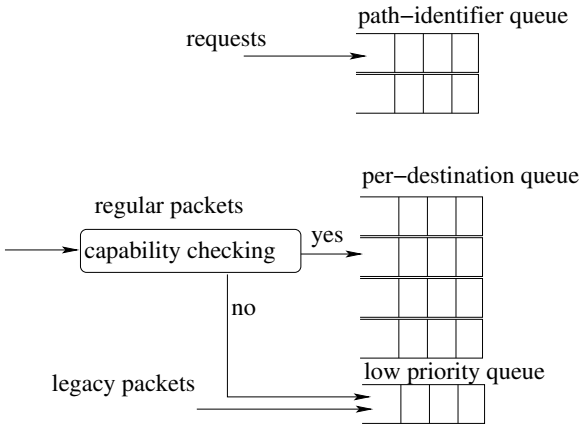

Figure 2: Queue management at a capability router. There are three types of traffic: requests that are rate-limited; regular packets with associated capabilities that receive preferential forwarding; and legacy traffic that competes for any remaining bandwidth.

an identifier for a network path. We then fair-queue requests using the most recent tag to identify a queue. This is shown in Figure 2. As a better approximation, earlier tags can be used within a given queue to break ties.

Queuing based on a path identifier has two benefits. First the number of queues is bounded because the tag space is bounded. This in turn bounds the complexity of fair queuing, ensuring that we will not exhaust router resources. Second, the scheme offers defense-in-depth because each trust domain such as an AS places the most trust in the domain that precedes it. Therefore, an attacker at the edge of the network or a compromised router who writes arbitrary tags can at most cause queue contention at the next downstream trust domain (AS). One consequence of this is that senders that share the same path identifier share fate, localizing the impact of an attack and providing an incentive for improved local security.

3.3 Destination Policies

The next question we consider is how a destination can determine whether to authorize a request. This is a matter of policy, and it depends on the role the destination plays in the network. We consider two extreme cases of a client and a public server to argue that simple policies can be effective.

A client may act in a way that by default allows it to contact any server but not otherwise be contacted, as is done by firewalls and NAT boxes today. To do this, it accepts incoming requests if they match outgoing requests it has already made and refuses them otherwise. Note that the client can readily do this because capabilities are added to existing packets rather than carried as separate packets. For example, a client can accept a request on a TCP SYN/ACK that matches its earlier request on a TCP SYN.

A public server may initially grant all requests with a default number of bytes and timeout, using the path identifier to fairly serve different sources when the load is high. If any of the senders misbehave, by sending unexpected packets or floods, that sender can be temporarily blacklisted and its capability will soon expire. This blacklisting is possible because the handshake involved in the capability exchange weakly authenticates that the source address corresponds to a real host. The result is that misbehaving senders are quickly contained. More sophisticated policies may be based on HTTP cookies that identify returning customers, CAPTCHAs that distinguish zombies from real users [11], and so forth.

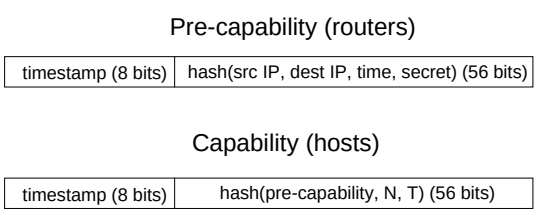

Figure 3: Format of capabilities.

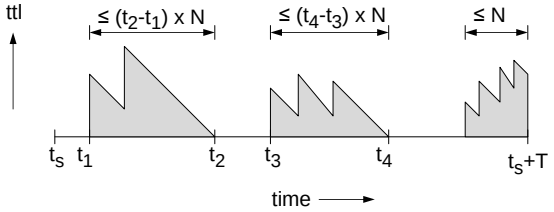

Figure 4: Bound on the bytes of a capability with caching.

3.4 Unforgeable Capabilities

Having provided a bootstrap mechanism and policy, we turn our attention to the form of capabilities themselves. Our key requirement is that an attacker can neither forge a capability, nor make use of a capability that they steal or transfer from another party. We also need capabilities to expire.

We use cryptography to bind each capability to a specific network path, including source and destination IP addresses, at a specific time. Each router that forwards a request packet generates its own pre-capability and attaches it to the packet. This pre-capability is shown in Figure 3. It consists of a local router timestamp and a cryptographic hash of that timestamp plus the source and destination IP addresses and a slowly-changing secret known only to the router. Observe that each router can verify for itself that a purported pre-capability attached to a packet is valid by re-computing the hash, since the router knows all of the inputs, but it is cryptographically hard for other parties to forge the pre-capability without knowing the router secret. Each router changes its secret at twice the rate of the timestamp rollover, and only uses the current or the previous secret to validate capability. This ensures that a pre-capability expires within at most the timestamp rollover period, and each pre-capability is valid for about the same time period regardless of when it is issued. The high-order bit of the timestamp indicates whether the current or the previous router secret should be used for validation. This trick allows a router to try only one secret even if the router changed its secret right after issuing a pre-capability.

The destination thus receives an ordered list of pre-capabilities that corresponds to a specific network path with fixed source and destination IP endpoints. It is this correspondence that prevents an attacker from successfully using capabilities issued to another party: it cannot generally arrange to send packets with a specific source and destination IP address through a specific sequence of routers unless it is co-located with the source. In the latter case, the attacker is indistinguishable from the source as far as the network is concerned, and shares its fate in the same manner as for requests. (And other, more devastating attacks are possible if local security is breached.) Thus we reduce remote exploitation to the problem of local security.

If the destination wishes to authorize the request, it returns an ordered list of capabilities to the sender via a packet sent in the reverse direction. Conceptually, the pre-capabilities we have described could directly serve as these capabilities. However, we process them further to provide greater control, as is described next.

3.5 Fine-Grained Capabilities

Even effective policies will sometimes make the wrong decision and the receiver will authorize traffic that ultimately is not wanted. For example, with our blacklist server policy an attacker will be authorized at least once, and with our client policy the server that a client accesses may prove to be malicious. If authorizations were binary, attackers whose requests were granted would be able to arbitrarily flood the destination until their capabilities expire. This problem would allow even a very small rate of false authorizations to deny service. This argues for a very short expiration period, yet protocol dynamics such as TCP timeouts place a lower bound on what is reasonable.

To tackle this problem, we design fine-grained capabilities that grant the right to send up to N bytes along a path within the next T seconds, e.g., 100KB in 10 seconds[2]. That is, we limit the amount of data as well as the period of validity. The form of these capabilities is shown in Figure 3. The destination converts the pre-capabilities it receives from routers to full capabilities by hashing them with N and T. Each destination can choose N and T (within limits) for each request, using any method from simple defaults to models of prior behavior. It is these full capabilities, along with N and T, that are returned to authorize the sender. For longer flows, the sender should renew these capabilities before they reach their limits.

With this scheme, routers verify their portion of the capabilities by re-computing the hashes much as before, except that now two hashes are required instead of one. The routers now perform two further checks, one for N and one for T. First, routers check that their local time is no greater than the router timestamp plus T to ensure that the capability has not expired. This requires that T be at most one half of the largest router timestamp so that two time values can be unambiguously compared under a modulo clock. The replay of very old capabilities for which the local router clock has wrapped are handled as before by periodically changing the router secret. Second, routers check that the capability will not be used for more than N bytes. This check is conceptually simple, but it requires state and raises the concern that attackers may exhaust router state. We deal with this concern next.

3.6 Bounded Router State

We wish to ensure that attackers cannot exhaust router memory to bypass capability limits. This is especially a concern given that we are counting the bytes sent with a capability and colluding attackers may create many authorized connections across a target link.

To handle this problem, we design an algorithm that bounds the bytes sent using a capability while using only a fixed amount of router state no matter how attackers behave. In the worst case, a capability may be used to send $2N$ bytes in T seconds. The same capability will still be precisely limited to N bytes if there is no memory pressure.

The high level idea of the algorithm is to make a router keep state only for flows (a flow is defined on a sender to a destination basis.) with valid capabilities that send faster than N/T. The router does not need to keep state for other authorized flows because they will

[2]An alternative would be to build rapid capability revocation. We believe this to be a less tractable problem.

not send more than N bytes before their capabilities expire in T seconds. We track flows via their rates by using the rate N/T to convert bytes to equivalent units of time, as we describe next.

When a router receives a packet with a valid capability for which it does not have state, it begins to track byte counts for the capability and also associates a minimal time-to-live (ttl) with the state. The ttl is set to the time equivalent value of the packet: $L * T/N$ seconds (with L being the packet length). This ttl is decremented as time passes (but our implementation simply sets an expiration time of $now + ttl$) and incremented as subsequent packets are charged to the capability. When the ttl reaches zero, it is permissible for the router to reclaim the state for use with a new capability.

We now show that this scheme bounds the number of bytes sent using a capability. Referring to Figure 4, suppose that the router created the capability at time t_s and it expires at time $t_s + T$. Further suppose that the router creates state for the capability at time $t_1 > t_s$, and reclaims the state when its ttl reaches zero at time $t_2 < t_s + T$. Then by the definition of the ttl, the capability must have been used for at most $(t_2 - t_1)/T * N$ bytes from t_1 to t_2. This may occur more than once, but regardless of how many times it occurs, the time intervals can total to no more than T seconds. Thus the total bytes used for the capability must be at most $T/T * N = N$ bytes. If a capability has state created at time immediately preceding $t_s + T$, then up to N bytes can be sent at a rate faster than N/T. Therefore, at most $N + N = 2N$ bytes can be sent before the capability is expired.

This scheme requires only fixed memory to avoid reclaiming state with non-zero ttl values, as required above. Suppose the capacity of the input link is C. To have state at time t, a capability must be used to send faster than N/T before t. Otherwise, the ttl associated with the state will reach zero and the state may be reclaimed. There can be at most $C/(N/T)$ such capabilities. We require that the minimum N/T rate be greater than an architectural constraint $(N/T)_{min}$. This bounds the state a router needs to $C/(N/T)_{min}$ records. As an example, if the minimum sending rate is 4K bytes in 10 seconds, a router with a gigabit input line will only need 312,500 records. If each record requires 100 bytes, then a line card with 32MB of memory will never run out of state.

3.7 Efficient Capabilities

We want capabilities to be bandwidth efficient as well as secure. Yet these properties are in conflict, since security benefits from long capabilities (i.e., a long key length) while efficiency benefits from short ones (i.e., less overhead). To reconcile these factors, we observe that most bytes reside in long flows for which the same capability is used repeatedly on packets of the flow. Thus we use long capabilities (64 bits per router) to ensure security, and cache capabilities at routers so that they can subsequently be omitted for bandwidth efficiency. We believe that this is a better tradeoff than short capabilities that are always present, e.g., SIFF uses 2 bits per router. Short capabilities are vulnerable to a brute force attack if the behavior of individual routers can be inferred, e.g., from bandwidth effects, and do not provide effective protection with a limited initial deployment.

In our design, when a sender obtains new capabilities from a receiver, it chooses a random flow nonce and includes it together with the list of capabilities in its packets. When a router receives a packet with a valid capability it caches the capability relevant information and flow nonce, and initializes a byte counter and ttl as previously described. Subsequent packets can then carry the flow nonce and omit the list of capabilities. Observe that path MTU discovery is likely unaffected because the larger packet is the first one sent to a destination. Routers look up a packet that omits its capabilities using its source and destination IP addresses, and compare the cached flow nonce with that in the packet. A match indicates that a router has validated the capabilities of the flow in previous packets. The packets are then subject to byte limit and expiration time checking as before.

For this scheme to work well, senders must know when routers will evict their capabilities from the cache. To do so, hosts model router cache eviction based on knowledge of the capability parameters and how many packets have used the capability and when. By the construction of our algorithm, eviction should be rare for high-rate flows, and it is only these flows that need to remain in cache to achieve overall bandwidth efficiency. This modeling can either be conservative, based on later reverse path knowledge of which packets reached the destination[3], or optimistic, assuming that loss is infrequent. In the occasional case that routers do not have the needed capabilities in cache, the packets will be demoted to legacy packets rather than lost, as we describe next.

3.8 Route Changes and Failures

To be robust, our design must accommodate route changes and failures such as router restarts. The difficulty this presents is that a packet may arrive at a router that has no associated capability state, either because none was set up or because the cache state or router secret has been lost.

This situation should be infrequent, but we can still minimize its disruption. First, we demote such packets to be the same priority as legacy traffic (which have no associated capabilities) by changing a bit in the capability header. They are likely to reach the destination in normal operation when there is little congestion. The destination then echoes demotion events to the sender by setting a bit in the capability header of the next message sent on the reverse channel. This tells the sender that it must re-acquire capabilities.

3.9 Balancing Authorized Traffic

Capabilities ensure that only authorized traffic will compete for the bandwidth to reach a destination, but we remain vulnerable to floods of authorized traffic: a pair of colluding attackers can authorize high-rate transfers between themselves and disrupt other authorized traffic that shares the bottleneck. This would allow, for example, a compromised insider to authorize floods on an access link by outside attackers.

We must arbitrate between authorized traffic to mitigate this attack. Since we do not know which authorized flows are malicious, if any, we simply seek to give each capability a reasonable share of the network bandwidth. To do this we use fair-queuing based on the authorizing destination IP address. This is shown in Figure 2. Users will now get a decreasing share of bandwidth as the network becomes busier in terms of users (either due to legitimate usage or colluding attackers), but they will be little affected unless the number of attackers is much larger than the number of legitimate users.

Note that we could queue on the source address (if source address can be trusted) or other flow definitions involving prefixes. The best choice is a matter of AS policy that likely depends on whether the source or destination is a direct customer of the AS, e.g., the source might be used when the packet is in the sender ISP's network and vice versa.

One important consideration is that we limit the number of queues to bound the implementation complexity of fair queuing. To do this, we again fall back on our router state bound, and fair-queue over the flows that have their capabilities in cache. In this manner, the high-rate flows that send more rapidly than N/T will fairly

[3]We ignore for the present the layering issues involved in using transport knowledge instead of building more mechanism.

share the bandwidth. These are the flows that we care most about limiting. The low-rate flows will effectively receive FIFO service with drops depending on the timing of arrivals. This does not guarantee fairness but is adequate in that it prevents starvation. An alternative approach would have been to hash the flows to a fixed number of queues in the manner of stochastic fair queuing [23]. However, we believe our scheme has the potential to prevent attackers from using deliberate hash collisions to crowd out legitimate users.

3.10 Short, Slow or Asymmetric Flows

TVA is designed to run with low overhead for long, fast flows that have a reverse channel. Short or slow connections will experience a higher relative overhead, and in the extreme may require a capability exchange for each packet. However, several factors suggest that TVA is workable even in this regime. First, the effect on aggregate efficiency is likely to be small given that most bytes belong to long flows. Second, and perhaps more importantly, our design does not introduce added latency in the form of handshakes, because capabilities are carried on existing packets, e.g., a request may be bundled with a TCP SYN and the capability returned on the TCP SYN/ACK. Third, short flows are less likely because flows are defined on a sender to a destination IP address basis. Thus all TCP connections or DNS exchanges between a pair of hosts can take place using a single capability.

TVA will have its lowest relative efficiency when all flows near a host are short, e.g., at the root DNS servers. Here, the portion of request bandwidth must be increased. TVA will then provide benefits by fair-queuing requests from different regions of the network. Truly unidirectional flows would also require capability-only packets in the reverse direction. Fortunately, even media streaming protocols typically use some reverse channel communications. Finally, we have not addressed IP multicast as it already require some form of authorization action from the receiver. It would be interesting to see whether we can provide a stronger protection in this setting by using capabilities.

4. TVA PROTOCOL

In this section, we describe TVA in terms of how hosts and routers process packets and provide a more detailed view of the common case for data transfer. We consider attacks more systematically in the following sections. We ignore legacy concerns for the moment, returning to them in Section 8.

There are three elements in our protocol: packets that carry capability information; hosts that act as senders and destinations; and routers that process capability information. We describe each in turn.

4.1 Packets with Capabilities

Other than legacy traffic, all packets carry a capability header that extends the behavior of IP. We implement this as a shim layer above IP, piggybacking capability information on normal packets so that there are no separate capability packets.

There are two types of packets from the standpoint of capabilities: request packets and regular packets. They share an identifying capability header and are shown in Figure 5. Request packets carry a list of blank capabilities and path identifiers that are filled in by routers as requests travel towards destinations. Regular packets have two formats: packets that carry both a flow nonce and a list of valid capabilities, and packets that carry only a flow nonce. (Recall that a flow is defined by a source and a destination IP address.) A regular packet with a list of capabilities may be used to request a new set of capabilities. We refer to such packets as renewal packets. If a regular packet does not pass the capability check, it may be

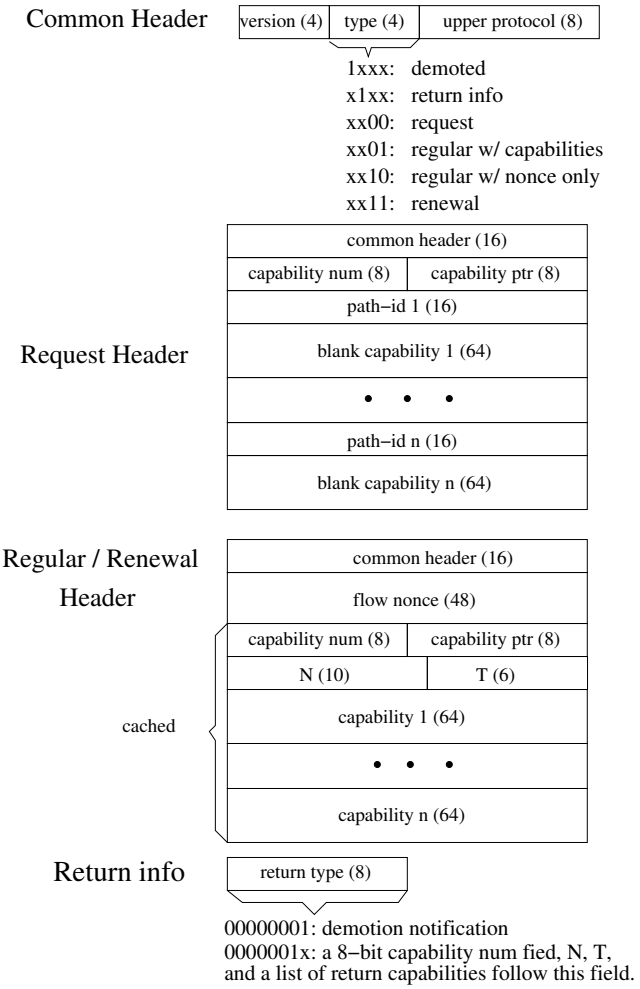

Figure 5: Types of capability packets. Return information is present if the return bit in the common header is set. Sizes are in bits. The units for N are KB; the units for T are seconds.

demoted to low priority traffic that is treated as legacy traffic. Such packets are called demoted packets.

We use the lowest two bits of the *type* field in the capability header to indicate the type and the format of packets: request packet, regular packet with a flow nonce only, regular packet with both a flow nonce and a list of capabilities, and renewal packet. One bit in the *type* field is used by routers to indicate that the packet has been demoted. The remaining bit indicates whether there is also return information being carried in the reverse direction to a sender. This information follows the capability payload. It may be a list of capabilities granted by the destination or a demote notification.

Each capability is as described in Section 3: a 64 bit value, broken down into 8 bits of router timestamp in seconds (a modulo 256 clock), and 56 bits of a keyed hash.

4.2 Senders and Destinations

To send to a destination for which it has no valid capabilities, a sender must first send a request. A request will typically be combined with the first packet a sender sends, such as a TCP SYN. When a destination receives the request, it must decide whether to grant or refuse the transfer. We described some simple policies in Section 3.3; there is also an issue we have not tackled of how to

express policies within the socket API. If the destination chooses to authorize the transfer, it sends a response with capabilities back to the sender, again combined with another packet, such as a TCP SYN/ACK. This SYN/ACK will also carry a request for the reverse direction. The reverse setup occurs in exactly the same manner as the forward setup, and we omit its description. To refuse the transfer, the destination may instead return an empty capability list, again combined with a packet such as a TCP RST.

Once the sender receives capabilities, the remainder of the transfer is straightforward. The sender sends data packets, initially with capabilities, and models capability expiration and cache expiration at routers to conservatively determine when routers will have their capabilities in cache, and when to renew the capabilities. In the common case, the flow nonce and capabilities are cached at every router. This enables the source to transmit most packets with only the flow nonce.

The destination simply implements a capability granting policy and does not need to model router behavior. It also echoes any demote signals to the sender, so that the sender may repair the path.

4.3 Routers

Routers route and forward packets as required by IP and additionally process packets according to the capability information that they carry. At a high level, routers share the capacity of each outgoing link between three classes of traffic. This is shown in Figure 2. Request packets, which do not have valid capabilities, are guaranteed access to a small, fixed fraction of the link (5% is our default) and are rate-limited not to exceed this amount. Regular packets with associated capabilities may use the remainder of the capacity. Legacy traffic is treated as the lowest priority, obtaining bandwidth that is not needed for either requests or regular packets in the traditional FIFO manner.

To process a request, the router adds a pre-capability to the end of the list and adds a new path identifier if it is at a trust boundary. The pre-capability is computed as the local timestamp concatenated with the hash of a router secret, the current, local router time in seconds using its modulo 256 clock, and the source and destination IP addresses of the packet. This is shown in Figure 3. The path identifier is a constant that identifies the ingress to the trust domain, either with high likelihood using pseudo-random functions or with configuration information. Requests are fair-queued for onward transmission using the most recent path identifiers.

To process a regular packet, routers check that the packet is authorized, update the cached information and packet as needed, and schedule the packet for forwarding. First, the router tries to locate an entry for the flow using the source and the destination IP address from the packet. An entry will exist if the router has received a valid regular packet from that flow in the recent past. The cache entry stores the valid capability, the flow nonce, the authorized bytes to send (N), the valid time (T), and the *ttl* and byte count as described in Section 3.6.

If there is a cached entry for the flow, the router compares the flow nonce to the packet. If there is a match, it further checks and updates the byte count and the *ttl*, and then fair queues the packet as described below. If the flow nonce does not match and a list of capabilities are present, this could be the first packet with a renewed capability, and so the capability is checked and if valid, replaced in the cache entry. Equivalently, if there is not a cached entry for the flow, the capability is checked, and a cache entry is allocated if it is valid. If the packet has a valid capability and is a renewal packet, a fresh pre-capability is minted and placed in the packet.

A router validates capability using the information in the packet (the source and destination addresses, N, and T) plus the router's secret. It recomputes the two hash functions to check whether they match the capability value. The router also checks that the byte count does not exceed N, and the current time does not exceed the expiration time (of timestamp $+T$) and updates the entry's ttl. Any packet with a valid capability or flow nonce is scheduled using fair queuing. Our scheme does this across flows cached at the router using destination addresses by default.

If neither the packet's flow nonce nor capability is valid, then the packet is marked as demoted and queued along with legacy packets. Figure 6 shows the pseudo-code on how a capability router processes a packet.

```
if (pkt->protocol == TVA) {
  isValid = false;
  if (isRTS(pkt)) { /* rts pkt */
    insert_precap_pi(pkt);
    enqueRts(pkt); /* per path identifier queue */
  } else  { /* regular pkt */
    entry = lookup(pkt);
    if (entry) { /* has entry */
      if (pkt->nonce == entry->nonce) {
        /* check byte count, expiration, update entry */
        isValid = updateEntry(entry, pkt);
      } else if (validateCap(pkt)) { /* comp two hashes */
        /* first renewed pkt. replace and check entry */
        isValid = replaceEntry(entry, pkt);
      }
    } else { /* no entry */
      if (validateCap(pkt)) {
        isValid = createEntry(pkt); /* create and check entry */
      }
    }
    if (isValid) {
      if (isRenewal(pkt)) { /* renewal pkt */
        renewPkt(pkt); /* insert precap */
      }
      enqueRegular(pkt); /* per-destination queue */
    } else {
      demote(pkt);
      enqueLegacy(pkt);
    }
  }
} else {
    enqueLegacy(pkt);
}
```

Figure 6: How a capability router processes a packet.

5. SIMULATION RESULTS

In this section, we use *ns* to simulate TVA, SIFF, pushback and the legacy Internet to see how well TVA limits the impact of DoS floods. TVA is as described in the previous sections, except that we rate-limit capability requests to 1% of the link capacity, down from our default of 5%, to stress our design. SIFF is implemented as described in [25]. It treats capacity requests as legacy traffic, does not limit the number of times a capability is used to forward traffic, and does not balance authorized traffic sent to different destinations. Pushback is implemented as described in [16]. It recursively pushes destination-based network filters backwards across the incoming link that contributes most of the flood.

For each scheme, we set up fixed length transfers between legitimate users, and a destination under various attacks. We then measure: i) the average fraction of completed transfers, and ii) the average time of the transfers that complete. These metrics are useful because a successful DoS attack will cause heavy loss that will

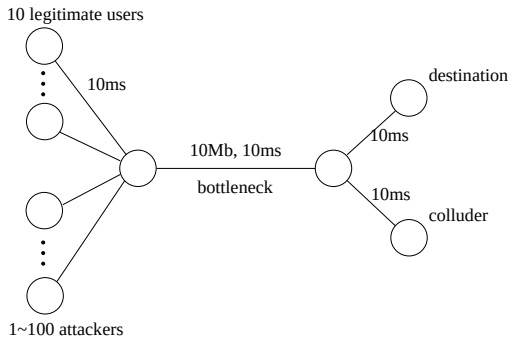

Figure 7: Simulation topology

both slow legitimate transfers and eventually cause the applications to abort them.

We simulate the dumbbell topology shown in Figure 7. The bottleneck link in our simulation is provisioned to give a legitimate user a nominal throughput of 1Mb/s over a bottleneck link with a nominal capacity of 10Mb/s. The RTT is 60ms. Each attacker floods at the rate of a legitimate user, 1Mb/s, and we vary intensity of the attacks from 1/10 of the bottleneck bandwidth to 10 times the bottleneck bandwidth by varying the number of attackers from 1 to 100. We use these relatively low rates to speed the simulation, since the key variables are the ratios between attacker, legitimate user, and the bottleneck bandwidth, given that there is a bandwidth-delay product sufficient to avoid small window effects.

Each legitimate user sends a 20KB file a thousand times using TCP, the next transfer starting after the previous one completes or aborts due to excessive loss. Capability requests are piggybacked on TCP SYNs. To provide a fair comparison for other schemes, we modify TCP to have a more aggressive connection establishment algorithm. Specifically, the timeout for TCP SYNs is fixed at one second (without the normal exponential backoff) and up to eight retransmissions are performed. Without this change, SIFF suffers disproportionately because it treats SYN packets with capability requests as legacy traffic, and therefore its performance under overload will be dominated by long TCP timeouts. Similarly, we set the TCP data exchange to abort the connection if its retransmission timeout for a regular data packet exceeds 64 seconds, or it has transmitted the same packet more than 10 times.

We note that TCP inefficiencies limit the effective throughput of a legitimate user to be no more than 533Kb/s in our scenario, given the transfer of 20KB with a 60ms RTT. This implies that there is virtually no bandwidth contention with a pool of 10 legitimate users – the contention effects we see come directly from massed attackers.

5.1 Legacy Packet Floods

The first scenario we consider is that of each attacker flooding the destination with legacy traffic at 1Mb/s. Figure 8 shows the fraction of completions and the average completion time for TVA in comparison with SIFF, pushback, and the current Internet.

We see that TVA maintains the fraction of completions near 100% and the average completion time remains small as the intensity of attacks increases. That is, our design strictly limits the impact of legacy traffic floods. This is because we treat legacy traffic with lower priority than request traffic.

SIFF treats both legacy and request packets as equally low priority traffic. Therefore, when the intensity of legacy traffic exceeds the bottleneck bandwidth, a legitimate user's request packets begin

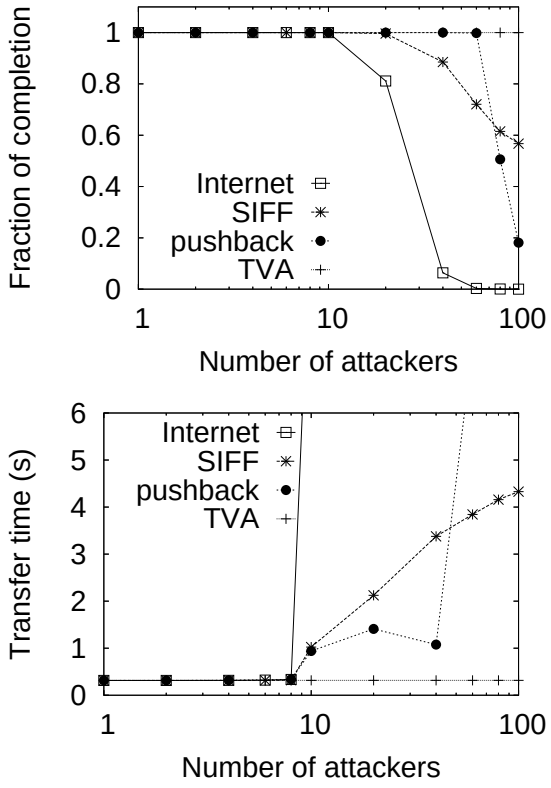

Figure 8: Legacy traffic flooding does not increase the file transfer time or decrease the fraction of completed transfers with TVA. With SIFF, file transfer time increases and the fraction of completed transfer drops when the intensity of attack increases; With pushback, as the number of attackers increases, pushback poorly isolates attack and legitimate traffic, and the transfer time increases and the fraction of completed transfer drops sharply; with today's Internet, the transfer time increases sharply, and the fraction of completion quickly approaches zero.

to suffer losses. We see this in Figure 8 when the number of attackers is greater than 10 and the fraction of completions drops and the completion time increases. When the aggregate attack bandwidth B_a is greater than the bottleneck bandwidth B_l, the packet loss rate p is approximately $(B_a - B_l)/B_a$. Once a request packet gets through, a sender's subsequent packets are authorized packets and are treated with higher priority. So the probability that a file transfer completes with SIFF is equal to the probability a request gets through within nine tries, i.e., $(1-p^9)$. When the number of attackers is 100, p is 90%, giving a completion rate of $(1-0.9^9) = 0.61$. This is consistent with our results. Similarly, the average time for a transfer to complete with up to 9 tries is:

$$T_{avg} = (\sum_{i=1}^{9} \cdot i \cdot p^{i-1} \cdot (1-p))/(1-p^9)$$

When there are 100 attackers, this time is 4.05 seconds, which is again consistent with our results.

Pushback performs well until the number of attackers is large, at which stage it provides poor isolation between attack traffic and legitimate traffic. This is because attack traffic becomes harder to identify as the number of attackers increases since each incoming link contributes a small fraction of the overall attack. As can be

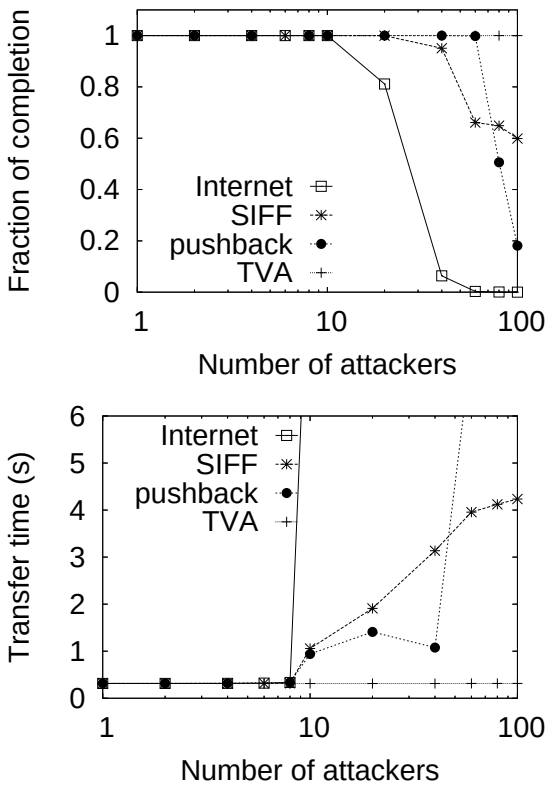

Figure 9: Request packet flooding does not increase the transfer time or decrease the fraction of completed transfers with TVA.

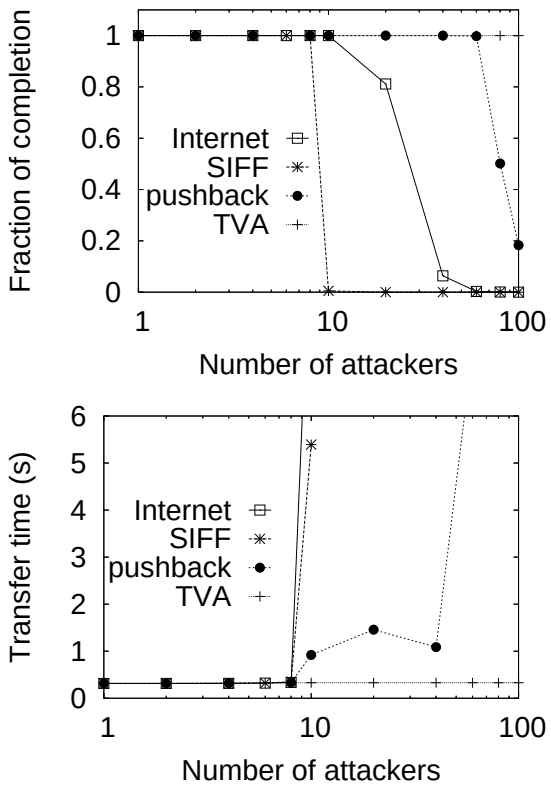

Figure 10: With TVA, per-destination queue ensures that the destination and the colluder equally share the access link bandwidth. The transfer time slightly increases (not noticeable from the figure) as a result of reduced bandwidth, and the fraction of completion remains 100%.

seen in Figure 8, when the number of attackers is less than 40, pushback is able to cut off a significant portion of the attack traffic: the file transfer time increases by less than 2 seconds, and the faction of completed transfers remains at 100%. However, pushback becomes markedly less effective as the number of attackers increases further. The file transfer time increases significantly, and the fraction of completed transfers drops sharply. TVA could be viewed as an alternative pushback scheme that is always activated and uses capabilities as a robust signature to separate attack traffic and legitimate traffic.

With the Internet, legitimate traffic and attack traffic are treated alike. Therefore, every packet from a legitimate user encounters a loss rate of p. The probability for a file transfer of n packets to get through, each within a fixed number of retransmissions k is $(1-p^k)^n$. This probability decreases polynomially as the drop rate p increases and exponentially as the number of packets n (or the file size) increases. This explains the results we see in Figure 8: the fraction of completed transfers quickly approaches to zero as the number of attackers increases. The few completed transfers have a completion time hundreds of seconds, and are out of the y-axis scope in Figure 8.

5.2 Request Packet Floods

The next scenario we consider is that of each attacker flooding the destination with request packets at 1Mb/s. In this attack, we assume the destination was able to distinguish requests from legitimate users and those from attackers. With TVA, request packets are rate limited and will not reduce the available capacity for authorized packets. Requests from attackers and legitimate users are queued separately, so that excessive requests from attackers will be dropped without causing requests from legitimate users to be dropped. We see this in Figure 9: the fraction of completions does not drop and the transfer time does not increase. In contrast, the results for SIFF are similar to those for legacy packets floods, as SIFF treats both requests and legacy traffic as low priority traffic. Both pushback and the legacy Internet treat RTS traffic as regular data traffic. The results for them are the same as those for the legacy traffic attack.

5.3 Authorized Packet Floods

Strategic attackers will realize that it is more effective to collude when paths can be found that share the bottleneck link with the destination. The colluders grant capabilities to requests from attackers, allowing the attackers to send authorized traffic at their maximum rate. Figure 10 shows the results under this attack. Because TVA allocates bandwidth approximately fairly among all destinations and allows destinations to use fine-grained capabilities to control how much bandwidth to allocate to a sender, this attack causes bandwidth to be fairly allocated between the colluder and the destination. The transfer time slightly increases (from 0.31 second to 0.33 second, not noticeable from the figure) as a result of reduced bandwidth, and all transfers complete. If the number of colluders that share a bottleneck link with the destination increases, the destination gets a decreased share of the bandwidth. Each legitimate user will get a lesser share of the bandwidth, but no user will be starved.

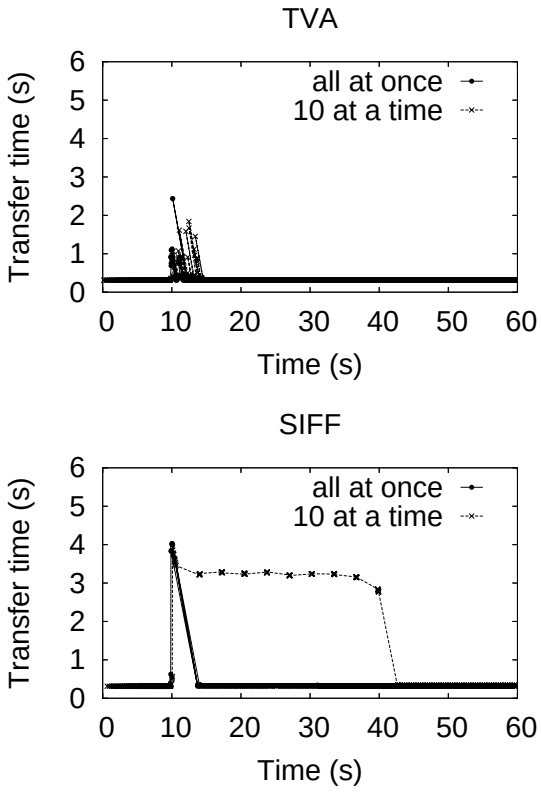

Figure 11: Attackers can only cause temporary damage if a destination stops renewing their capabilities. TVA uses a fine-grained capability to limit the impact of authorizing an attacker to a smaller amount of attack traffic compared to SIFF, even assuming SIFF has a rapid-changing router secret that expires every 3 seconds.

Under the same attack with SIFF, legitimate users are completely starved when the intensity of the attack exceeds the bottleneck bandwidth. Again, this is because the request packets are treated with low priority and are dropped in favor of the authorized attack traffic. We see in Figure 10 that the request completion rate drops sharply to zero when the attacking bandwidth reaches the bottleneck bandwidth of 10Mb/s. The very few transfers that complete do so only because of traffic fluctuations, and suffer a sharp increase in the average transfer time.

Both pushback and the legacy Internet treat request traffic and authorized traffic as regular traffic. Thus, the results for each scheme under an authorized traffic attack is similar to each scheme under a legacy traffic attack. Pushback will identify the flow aggregate destined to the colluder as the offending aggregate that causes most packet drops, but it fails to rate limit the aggregate to its fair share of bandwidth.

5.4 Imprecise Authorization Policies

Finally, we consider the impact of imprecise policies, when a destination sometimes authorizes attackers because it cannot reliably distinguish between legitimate users and attackers at the time that it receives a request. In the extreme that the destination cannot differentiate attackers from users at all, it must grant them equally.

However, if the destination is able to differentiate likely attack requests, even imprecisely, TVA is still able to limit the damage of DoS floods. To see this, we simulate the simple authorization policy described in Section 3.3: a destination initially grants all requests, but stops renewing capabilities for senders that misbehave by flooding traffic. We set the destination to grant an initial capability of 32KB in 10 seconds. This allows an attacker to flood at a rate of 1Mb/s, but for only 32KB until the capability expires. The destination does not renew capabilities because of the attack. Figure 11 shows how the transfer time changes for TVA with this policy as an attack commences. There are two attacks: a high intensity one in which all 100 attackers attack simultaneously; and a low intensity one in which the 100 attackers divide into 10 groups that flood one after the other, as one group finishes their attack. We see that both attacks are effective for less than 5 seconds, causing temporary congestion and increasing the transfer time of some connections by about 2 seconds.

Figure 11 also shows the results for SIFF under the same attacks. In SIFF, the expiration of a capability depends on changing a router secret – even if the destination determines that the sender is misbehaving it is powerless to revoke the authorization beforehand. This suggests that rapid secret turnover is needed, but there are practical limitations on how quickly the secret can be changed, e.g., the life time of a router secret should be longer than a small multiple of TCP timeouts. In our experiment, we assume SIFF can expire its capabilities every three seconds. By contrast, TVA expires router secret every 128 seconds. We see that both attacks have a much more pronounced effect on SIFF. The high intensity attack increases the transfer time by 4 seconds, and the low intensity attack lasts for 30 seconds. In each attack period of three seconds, all legitimate requests are blocked until the next transition. As a result, the transfer time jumps to more than three seconds.

Packet type	Processing time
Request	460 ns
Regular with a cached entry	33 ns
Regular without a cached entry	1486 ns
Renewal with a cached entry	439 ns
Renewal without a cached entry	1821 ns

Table 1: Processing overhead of different types of packets.

6. IMPLEMENTATION

We prototyped TVA using the Linux netfilter framework [19] running on commodity hardware. We implemented the host portion of the protocol as a user-space proxy, as this allows legacy applications to run without modification. We implemented router capability processing as a kernel module using the AES-hash as the first hash function (for pre-capabilities) and SHA1 as the second hash function [17] (for capabilities).

The purpose of this effort was to check the completeness of our design and to understand the processing costs of capabilities. We did not consider the processing costs of fair queuing. In our experiment, we set up a router using a dual-processor 3.2GHz Pentium Xeon machine running a Linux 2.6.8 Kernel. It used the native Linux forwarding module. We then used a kernel packet generator to generate different types of packets and sent them through the router, modifying the code to force the desired execution path. For each run, our load generator machines sent one million packets of each type to the router. We recorded the average number of instruction cycles for the router to process each type of packet, averaging the results over five experiments.

Table 1 shows the results of this experiment, with cycles converted to time. In normal operation, the most common type of packet is a regular packet with an entry at a router. The processing

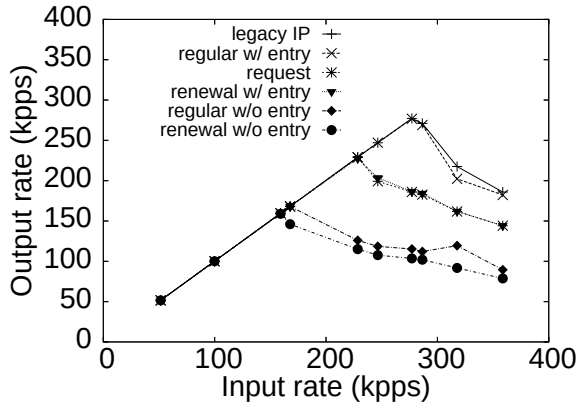

Figure 12: The peak output rate of different types of packets.

overhead for this type is the lowest at 33 ns. The processing overhead for validating a capability for a packet without a cached entry is about 1486 ns, as it involves computing two hash functions. The cost to process a request packet is lower and similar to the cost to process a renewal packet with a cached entry because both involve a pre-capability hash computation. The most computation-intensive operation is forwarding a renewal packet without a cached entry. In this case the router needs to compute three hash functions: two to check the validity of the old capability, and one to compute a new pre-capability hash. The processing cost is 1821 ns.

We also tested how rapidly a Linux router could forward capability packets. The results are shown in Figure 12. The output rate increases with the input rate and reaches a peak of 160 to 280Kpps, depending on the type of packet. This compares well with the peak lossless rate for vanilla IP packets of about 280Kpps. In both cases these rates are dominated by per packet interrupt handling, and they could be increased markedly with a polling device driver, as demonstrated by Click [13]. We expect that removing the 3.5us interrupt penalty would improve the output rate to 500-1400Kpps, equivalent to 240 to 670Mbps with minimum size packets (of 40 TCP/IP bytes plus 20 capability bytes). An attacker might attempt to overwhelm the CPU by flooding spoofed short renewal packets; they would not match, but that might still lead to packet loss of good traffic if the processing was done in the order received. Fortunately, we can use Lazy Receiver Processing (LRP) for this case [7]: when the CPU is overloaded, separately queue incoming packets based on their required computation per input bit. Normal traffic, consisting of short requests and full-size regular packets will then be processed at full speed.

We conclude that our implementation can handle 100 Mbps interfaces with off-the-shelf hardware; in the near future, we expect to be able to demonstrate that an optimized implementation can run at a gigabit without specialized hardware.

7. SECURITY ANALYSIS

The security of TVA is based on the inability of an attacker to obtain capabilities for routers along the path to a destination they seek to attack. We briefly analyze how TVA counters various threats.

An attacker might try to obtain capabilities by breaking the hashing scheme. We use standard cryptographic functions with a sufficient amount of key material and change keys every 128 seconds as to make breaking keys a practical impossibility.

An attacker may try to observe the pre-capabilities placed in its requests by routers, e.g., by causing ICMP error messages to be returned to the sender from within the network, or by using IP source routing. To defeat these vulnerabilities, we use a packet format that does not expose pre-capabilities in the first 8 bytes of the IP packet (which are visible in ICMP messages) and require that capability routers treat packets with IP source routes as legacy traffic. Beyond this, we rely on Internet routing to prevent the intentional misdelivery of packets sent to a remote destination.

A different attack is to steal and use capabilities belonging to a sender (maybe another attacker) who was authorized by the destination. Since a capability is bound to a specific source, destination, and router, the attacker will not generally be able to send packets along the same path as the authorized sender. The case in which we cannot prevent theft is when the attacker can eavesdrop on the traffic between an authorized sender and a destination. This includes a compromised router. In this case, the attacker can co-opt the authorization that belongs to the sender. In fact, it can speak for any senders for whom it forwards packets. However, even in this situation our design provides defense in depth. The compromised router is just another attacker – it does not gain more leverage than an attacker at the compromised location. DoS attacks on a destination will still be limited as long as there are other capability routers between the attacker and the destination.

Another attack an eavesdropper can launch is to masquerade a receiver to authorize attackers to send attack traffic to the receiver. Similarly, our design provides defense in depth. If the attacker is a compromised router, this attack can only congest the receiver's queues at upstream links, because the router cannot forge pre-capabilities of downstream routers. This attack is no worse than the router simply dropping all traffic to the receiver. If the attacker is a comprised host that shares a local broadcast network with a receiver, the attacker can be easily spotted and taken off-line.

Alternatively, an attacker and a colluder can spoof authorized traffic as if it were sent by a different sender S. The attacker sends requests to the colluder with S's address as the source address, and the colluder returns the list of capabilities to the attacker's real address. The attacker can then flood authorized traffic to the colluder using S's address. This attack is harmful if per-source queuing is used at a congested link. If the spoofed traffic and S's traffic share the congested link, S's traffic may be completely starved. This attack has little effect on a sender's traffic if per-destination queueing is used, which is TVA's default. ISPs should not use per-source queuing if source addresses cannot be trusted.

Finally, other attacks may target capability routers directly, seeking to exhaust their resources. However, the computation and state requirements for our capability are bounded by design. They may be provisioned for the worst case.

8. DEPLOYMENT

Our design requires both routers and hosts to be upgraded, but does not require a flag day. We expect incremental deployment to proceed organization by organization. For example, a government or large scale enterprise might deploy the system across their internal network, to ensure continued operation of the network even if the attacker has compromised some nodes internal to the organization, e.g., with a virus. Upstream ISPs in turn might deploy the system to protect communication between key customers.

Routers can be upgraded incrementally, at trust boundaries and locations of congestion, i.e., the ingress and egress of edge ISPs. This can be accomplished by placing an inline packet processing box adjacent to the legacy router and preceding a step-down in capacity (so that its queuing has an effect). No cross-provider or inter-router arrangements are needed and routing is not altered. Further deployment working back from a destination then provides greater

protection to the destination in the form of better attack localization, because floods are intercepted earlier.

Hosts must also be upgraded. We envision this occurring with proxies at the edges of customer networks in the manner of a NAT box or firewall. This provides a simpler option than upgrading individual hosts and is possible since legacy applications do not need to be upgraded. Observe that legacy hosts can communicate with one another unchanged during this deployment because legacy traffic passes through capability routers, albeit at low priority. However, we must discover which hosts are upgraded if we are to use capabilities when possible and fall back to legacy traffic otherwise. We expect to use DNS to signal which hosts can handle capabilities in the same manner as other upgrades. Additionally, a capability-enabled host can try to contact a destination using capabilities directly. This will either succeed, or an ICMP protocol error will be returned when the shim capability layer cannot be processed, as evidence that the host has not been upgraded.

9. CONCLUSION

We have presented and evaluated TVA, a network architecture that limits denial of service attacks so that two hosts are able to communicate effectively despite a large number of attackers; we have argued that existing mechanisms do not meet this goal. Our design is based on the concept of capabilities that enable destinations to authorize senders, in combination with routers that preferentially forward authorized traffic. Our main contribution is to flesh out the design of a comprehensive and practical capability system for the first time. This includes protections for the initial request exchange, consideration of destination policies for authorizing senders, and ways to bound both router computation and state requirements. Our simulation results show that, with our design, even substantial (10x) floods of legacy traffic, request traffic, and other authorized traffic have little or limited impact on the performance of legitimate users. We have striven to keep our design practical. We implemented a prototype of our design in the Linux kernel, and used it to argue that our design will be able to run at gigabit speeds on commodity PCs. We also constrained our design to be easy to transition into practice. This can be done by placing inline packet processing boxes near legacy routers, with incremental deployment providing incremental gain.

10. ACKNOWLEDGEMENTS

We thank Ratul Mahajan for help with the pushback approach, Ersin Uzun for pointing out the attack on per-source queuing, and the anonymous SIGCOMM reviewers for their comments. This work was supported in part by the NSF (Grant CNS-0430304).

11. REFERENCES

[1] D. Andersen. Mayday: Distributed Filtering for Internet Services. In *3rd Usenix USITS*, 2003.

[2] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet Denial of Service with Capabilities. In *Proc. HotNets-II*, Nov. 2003.

[3] K. Argyraki and D. Cheriton. Active Internet Traffic Filtering: Real-Time Response to Denial-of-Service Attacks. In *USENIX 2005*, 2005.

[4] DDoS attacks still pose threat to Internet. BizReport, 11/4/03.

[5] Extortion via DDoS on the rise. Network World, 5/16/05.

[6] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *ACM SIGCOMM*, 1989.

[7] P. Druschel and G. Banga. Lazy Receiver Processing (LRP): A Network Subsystem Architecture for Server Systems. In *2nd OSDI*, 1996.

[8] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks that Employ IP Source Address Spoofing. Internet RFC 2827, 2000.

[9] M. Handley and A. Greenhalgh. Steps Towards a DoS-Resistant Internet Architecture. In *ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, 2004.

[10] J. Ioannidis and S. Bellovin. Implementing Pushback: Router-Based Defense Against DoS Attacks. In *NDSS*, 2002.

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

[12] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *ACM SIGCOMM*, 2002.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[14] K. Lakshminarayanan, D. Adkins, A. Perrig, and I. Stoica. Taming IP Packet Flooding Attacks. In *Proc. HotNets-II*, 2003.

[15] S. Machiraju, M. Seshadri, and I. Stoica. A Scalable and Robust Solution for Bandwidth Allocation . In *IWQoS'02*, 2002.

[16] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling High Bandwidth Aggregates in the Network. *Computer Communications Review*, 32(3), July 2002.

[17] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*, chapter 9. CRC Pres, 1997.

[18] D. Moore, G. Voelker, and S. Savage. Inferring Internet Denial of Service Activity. In *Usenix Security Symposium 2001*, 2001.

[19] `http://www.netfilter.org/`.

[20] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *ACM SIGCOMM*, 2000.

[21] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, S. Kent, and W. Strayer. Hash-Based IP Traceback. In *ACM SIGCOMM*, 2001.

[22] D. Song and A. Perrig. Advance and Authenticated Marking Schemes for IP Traceback. In *Proc. IEEE Infocom*, 2001.

[23] I. Stoica, S. Shenker, and H. Zhang. Core-Stateless Fair Queueing: Achieving Approximately Fair Bandwidth Allocations in High Speed Networks. In *ACM SIGCOMM*, 1998.

[24] A. Yaar, A. Perrig, and D. Song. Pi: A Path Identification Mechanism to Defend Against DDoS Attacks. In *IEEE Symposium on Security and Privacy*, 2003.

[25] A. Yaar, A. Perrig, and D. Song. SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks. In *IEEE Symposium on Security and Privacy*, 2004.