TVA: a DoS-limiting Network Architecture

Xiaowei Yang, *Member,* David Wetherall, *Member,* Thomas Anderson, *Member*

***Abstract*— We motivate the capability approach to network denial-of-service (DoS) attacks, and evaluate the TVA architecture which builds on capabilities. With our approach, rather than send packets to any destination at any time, senders must first obtain "permission to send" from the receiver, which provides the permission in the form of capabilities to those senders whose traffic it agrees to accept. The senders then include these capabilities in packets. This enables verification points distributed around the network to check that traffic has been authorized by the receiver and the path in between, and hence to cleanly discard unauthorized traffic. To evaluate this approach, and to understand the detailed operation of capabilities, we developed a network architecture called TVA. TVA addresses a wide range of possible attacks against communication between pairs of hosts, including spoofed packet floods, network and host bottlenecks, and router state exhaustion. We use simulations to show the effectiveness of TVA at limiting DoS floods, and an implementation on Click router to evaluate the computational costs of TVA. We also discuss how to incrementally deploy TVA into practice.**

I. Introduction

The Internet owes much of its historic success and growth to its openness to new applications. A key design feature of the Internet is that any application can send anything to anyone at any time, without needing to obtain advance permission from network administrators. New applications can be designed, implemented and come into widespread use much more quickly, if they do not need to wait for key features to be added to the underlying network.

Quietly, however, the Internet has become much less open to new applications over the past few years. Perversely, this has happened as a rational response of network and system administrators needing to cope with the consequences of the Internet's openness. The Internet architecture is vulnerable to denial-of-service (DoS) attacks, where any collection of hosts with enough bandwidth (e.g., using machines taken over by a virus attack) can disrupt legitimate communication between any pair of other parties, simply by flooding one end or the other with unwanted traffic. These attacks are widespread, increasing, and have proven resistant to all attempts to stop them [26].

Operationally, to deal with persistent and repeated DoS and virus attacks, network and system administrators have begun to deploy automated response systems to look for anomalous behavior that might be an attack. When alarms are triggered, often by legitimate traffic, the operational response is typically to "stop everything and ask questions later." Unfortunately, any new application is likely to appear to be anomalous! Our experience with this comes from operating and using the PlanetLab testbed, which is designed to make it easy to develop new, geographically distributed, Internet applications [27]. On several occasions, we have observed innocuous, low-rate traffic from a single application trigger alarms that completely disconnected entire universities from the Internet. Since alarm rules are by nature secret, the only way to guarantee that a new application does not trigger an alarm (and the resulting disproportionate response) is to make its traffic look identical to some existing application. In other words, the only safe thing to do is to precisely mimic an old protocol.

The openness of the Internet is likely to erode if there is no effective solution to eliminate large scale DoS attacks. Attackers are winning the arms race with anomaly detection by making their traffic look increasingly like normal traffic. The CodeRed and follow-on viruses have demonstrated repeatedly that it is possible to recruit millions of machines to the task of sending normal HTTP requests to a single destination [24], [25]. This problem is fundamental to the Internet architecture: no matter how over-provisioned you are, if everyone in the world sends you a single packet, legitimate traffic will not get through.

We argue for taking a step back, to ask how, at an architectural level, we can address the DoS problem in its entirety while still allowing new applications to be deployed. Our goal, in essence, is to let any two nodes exchange whatever traffic they like (subject to bandwidth constraints of intermediate links), such that no set of third parties can disrupt that traffic exchange.

Our approach is based on the notion of capabilities, which are short-term authorizations that senders obtain from receivers and stamp on their packets. This allows senders to control the traffic that they receive. Our attraction to capabilities is that they cut to the heart of the DoS problem by allowing unwanted traffic to be removed in the network, but do so in an open manner by providing destinations with the control over which traffic is filtered. However, while capabilities may be an appealing approach, they leave many questions unanswered, such as how capabilities are granted without being vulnerable to attack.

To answer these questions and help evaluate the capability approach, we have designed and prototyped the Traffic Validation Architecture (TVA[1]). TVA is a DoS-limiting network architecture that details the operation of capabilities and combines mechanisms that counter a broad set of possible denial-of-service attacks, including those that flood the setup channel, that exhaust router state, that consume network bandwidth, and so forth. The design that we present in this paper is a revision of our earlier work [35] that pays greater attention to protecting the capability request channel.

We have designed TVA to be practical in three key respects. First, we bound both the computation and state needed to process capabilities. 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

Xiaowei Yang is with University of California at Irvine; David Wetherall is with both University of Washington and Intel Research Seattle. Thomas Anderson is with University of Washington. This work was supported in part by the NSF (Grant CNS-0430304 and Grant CNS-0627787).

[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.

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 motivate the capability approach in the context of related work in Section II. Section III and IV present a concrete design and implementation of a capability-based network architecture. Sections V, VI, and VII evaluate our approach using a combination of simulation, a Click router implementation, and analysis. Section VIII discusses TVA's deployment issues, and future directions. Section IX summarizes our work.

II. Background and Related Work

Early work in the area of DoS sought to make all sources identifiable, e.g., ingress filtering [12] 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 [28], [30], [31]. This is a key step, but it is insufficient as a complete solution, as attackers may still launch packet floods with unspoofed packets.

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 [18] and Mayday [2] take this approach. This approach does not protect public servers (e.g., www.google.com) that are in general unable to arrange an off-line authenticator for legitimate senders prior to communication.

Handley and Greenhalgh [13] propose to limit host communication patterns to client-server only by separating client and server address spaces. The proposal *Off by Default* [6] is similar in spirit. The network does not permit any two hosts to communicate by default, unless a destination explicitly requests to receive from a sender. Both solutions limit DoS attacks to private end hosts, but require additional mechanisms to protect open public servers.

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], [21] and more recently AITF [4]. 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.

Perhaps the most active area of DoS prevention work is anomaly detection [7], [15]. Rule-based or statistical techniques are used to classify traffic patterns as friendly or malicious. However, anomaly detection is not a sufficient response to the DoS problem—the decision as to whether a particular flow is an attack or not needs to be made end-to-end at the application level. Worse, in the limit anomaly detection leads to a closed Internet that stifles innovations, as ISPs and sysadmins lock down everything that isn't completely standard in the arms race with attackers.

Therefore, we propose the approach of putting a capability into each data packet to demonstrate that the packet was requested by the receiver in [3]. Communication takes two steps: 1) the sender requests permission to send; 2) after verifying the sender is good, the receiver provides it with a capability. When included in a packet, this capability 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 [3] 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 [34]. This is reminiscent of work in robust admission control [20]. Our design TVA adopts this approach, with some enhancements motivated by the 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. Given the timeout constants suggested in [34], 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.

III. TVA Design Overview

In this section, we motivate the key components of TVA. Later in Section IV, 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.

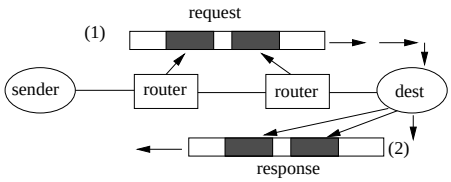


Fig. 1. **A sender obtaining initial capabilities by (1) sending a request to the destination, to which routers add pre-capabilities; and (2) receiving a response, to which the destination added capabilities.**

A. 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 [3].

Capabilities have significant potential benefits compared to other schemes that describe unwanted packets using implicit features [16], [21]. 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, capabilities 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.

B. 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 communication 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.

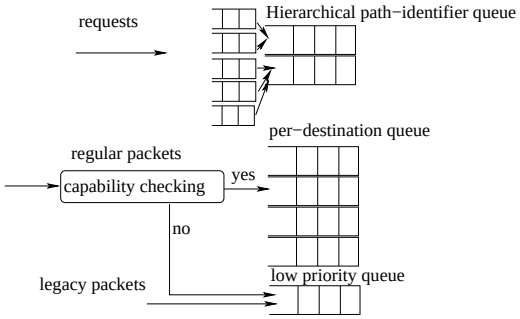


Fig. 2. **Queue management at a capability router. There are three types of traffic: requests that are rate-limited; regular packets with associated capabilities that receive preferential forwarding; and legacy traffic that competes for any remaining bandwidth.**

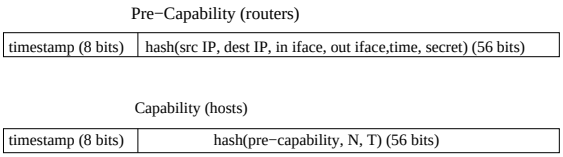


Fig. 3. **Format of capabilities.**

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 [33] 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 an identifier for a network path. We then hierarchically fair-queue [8] requests using path identifiers, as shown in Figure 2. The most recent tag is used to identify the first-level queue, and the second most recent tag is used to identify the second-level queue, and so on. If a queue at the $(n-1)$th-level is congested, a router will use the nth most recent tag to separate packets into nth-level queues. If

the number of queues a router can support is greater than the number of trust domains that use the router to reach a destination, and attackers do not insert faked path identifier tags, this queueing mechanism will effectively separate attackers' requests from legitimate requests, even if attackers fake their source addresses.

However, an attacker may attempt to exhaust a router's queues by inserting arbitrary path identifier tags in its request packets, and then flood those packets to congest the request queues. This may cause a router to create many queues to separate the faked path identifiers. Our design uses a queue balancing algorithm to limit the effect of this attack. If a $(n-1)$th-level queue is needed, but a router has reached its queue limit, the queue balancing algorithm would merge two queues at a deeper level (e.g., at the nth level) to a lower-level queue (e.g., to $n-1$th-level) to make space for the new queue. This algorithm prevents an attacker from grabbing an arbitrarily large number of queues by spoofing path identifiers. In the worst case that a router runs out of queues, legitimate users that are far away from a router are more likely to share queues with attackers close to them, localizing the impact of an attack.

This hierarchical queueing mechanism is a significant improvement over an earlier design of TVA [35], which fairly queues packets using the most recent tags rather than hierarchically fair-queue packets using all path identifier tags. If attackers and legitimate users share partial paths, requests from legitimate senders may be overwhelmed by requests from attackers.

Hierarchically queuing based on a path identifier has two benefits. First the number of queues is bounded to a router's preset queue limit even in the presence of source address or path identifier spoofing. Second, the scheme offers defense-in-depth because each trust domain such as an AS places the most trust in domains that are closest. The hierarchical queuing mechanism gives higher shares of a router's queues and correspondingly request channel bandwidth to request packets coming from domains that are closer, because it merges deepest queues first when a router hits its queue limit.

C. 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, but defer the study on optimal receiver policies for future study.

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 [10], [17], and so forth.

D. 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. Figure 3 shows this pre-capability. 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 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 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.

E. 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.

F. 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 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.

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

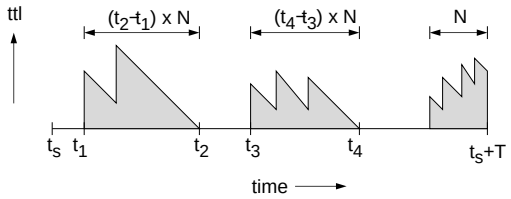


Fig. 4. **Bound on the bytes of a capability with caching.**

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. This amount of fast memory is not trivial, but appears modest.

G. 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 process is likely unaffected because the larger packet is the first one sent to a destination, but subsequent packets sent may be slightly smaller than MTU. 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.

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

I. 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 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 [22]. However, we believe our scheme has the potential to prevent attackers from using deliberate hash collisions to crowd out legitimate users.

J. 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 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 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

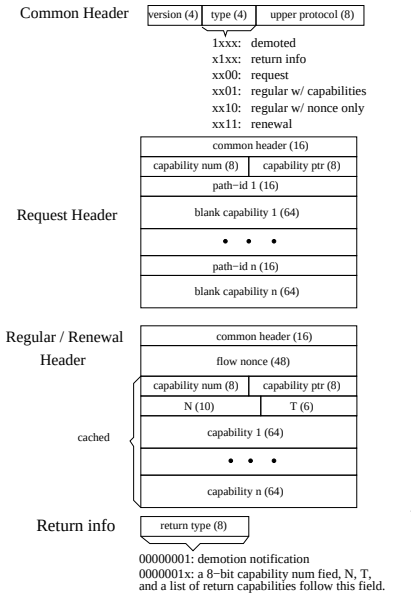


Fig. 5. **Types of capability packets. Return information is present if the return bit in the common header is set. Sizes are in bits. The units for N are KB; the units for T are seconds.**

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 requires some form of authorization from the receiver. It would be interesting to see whether we can provide a stronger protection in this setting by using capabilities.

IV. 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 VIII.

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.

A. 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 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.

B. 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 III-C; 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.

C. 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 III-F.

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.

V. Simulation Results

In this section, we use *ns-2* to simulate TVA to see how well it limits the impact of DoS floods. We compare TVA with SIFF, pushback, and the legacy Internet to highlight various design choices of TVA. TVA is implemented as described in the previous sections. Routers rate limit capability requests to 5% of the link capacity. SIFF is implemented as described in [34]. 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. We use the Pushback implementation described in [21]. It recursively pushes destination-based network filters backwards across the incoming link that contributes most of the flood.

We first describe our experimental methodology. Due to the complexity of Internet topologies and attacker strategies, it is a challenging task to design high-fidelity experiments to compare different DoS solutions. We make a best-effort attempt to base our experiments on realistic Internet topologies and estimated attacker strategies.

A. Methodology

Comparison metrics. For each scheme, we set up TCP file transfers between legitimate users and a destination under various attacks. We then measure the distribution of the file transfer times of legitimate users. This metric is useful because a successful DoS attack will cause heavy loss that will slow legitimate transfers and eventually cause the applications to abort them.

Topologies. Simulations of TVA require knowing the path identifier distribution of legitimate users and attackers seen at a bottleneck. Unfortunately, this information is not readily available. Instead, we approximate it using AS paths included in BGP dumps from the Oregon RouteView and RIPE RIS servers. The BGP dumps were obtained between April and May 2007. We use the reversed best AS path from a vantage point to an AS to approximate the forwarding path from that AS to the vantage point. We then generate AS-level network topologies using the AS path information. Each topology includes around 35K unique AS paths and 25K ASes.

Unfortunately, our simulator cannot simulate topologies at this scale. To address this issue, we partition the Internet-scale topology into sub-topologies using path identifier prefixes. For instance, suppose the vantage point AS tags a neighbor with an identifier p_i. Then all ASes with the path identifier prefix p_i* belong to the sub-topology p_i*. We then randomly sample the largest sub-topologies that our simulator can handle, ie., sub-topologies with 1000$\sim$2000 ASes. Intuitively, the larger a sub-topology is, the more similar it is to the original AS-level Internet topology. We simulated a total of six sub-topologies sampled from five different vantage points, and the results presented in this section are take from one representative sub-topology from the Oregon OIX vantage point. Other results are mostly similar, and are included in [1].

For each sub-topology, the bottleneck link lies between the AS that is closest to the vantage point and the vantage point. The victim destination and a colluder are behind the vantage point.

Parameters. For each sub-topology, we randomly mark $d\%$ of edge ASes as attackers, with d ranging from 10, 20, 40, to 80. Unmarked edge ASes are legitimate users. We also randomly mark 25% of edge ASes as spoofers. This number is set according to the Spoofer [9] project that shows close to 25% of ASes still allow address spoofing. We assume that ASes that do not allow address spoofing will not allow path spoofing were TVA deployed. In our simulations, an AS marked as an attacker sends packet floods. If an AS is marked both as an attacker and spoofer, it sends packet floods with spoofed path identifier tags.

Since BGP uses prefix-based route selection and an AS may announce multiple prefixes, there are multiple paths between two ASes. As ns-2 only supports single-path routing, we create

one node corresponding to one path identifier of an AS in ns-2. If an AS is marked as an attacker or spoofer in the marking process, all instances of the AS in the simulation topology are attackers or spoofers.

In our simulations, each attacker instance sends 1Mb/s traffic. The bottleneck bandwidth is set to one tenth of the aggregate attack bandwidth when the attacker density is 80%. The non-bottleneck links are set to 10Gb/s. Link delay is set between 5ms to 10ms. TVA's results also depend on the number of request queues a bottleneck router can handle. In our simulations, we assume 200K queues are available to an Internet-scale topology with N_I unique path identifiers. We choose 200K because our prototype implementation (Section VI) on a commodity PC can support this number. We scale the number of queues allocated to a sub-topology to $N_s/N_I * 200K$, where N_s is the number of path identifiers seen in the sub-topology. The maximum depth of a hierarchical queue is set to four based on our prototype implementation. This is because most AS path lengths are less than five, and the sampled sub-topologies are often one AS hop away from a vantage point. A few legitimate users that share the last four path identifier tags with attackers are not protected.

The TCP file transfer size in our simulations is 20KB. A new capability request is piggybacked on the TCP SYN packet of each transfer. We choose a small and fixed file size to speed up the simulations and for clarity: we use this sample point to explain the performance difference of different schemes. Although there is evidence that most TCP flows are less than 20KB [36], most bytes are sent by flows longer than 100KB [36]. Besides, as TVA's capabilities are requested on a per-host basis, multiple short flows (e.g., embedded images in a web page) only need to send one request. Thus we believe a transfer size of 20KB is a fair choice for our simulations, and the benefits of capabilities are more prominent for longer transfers. Capability processing overhead is not simulated, as it is evaluated in Section **??**. Capability cache misses are not simulated, because caching is an optimization, and a detailed cache eviction algorithm is left for further study.

To provide a fair comparison to 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). Without this change, SIFF suffers disproportionately because it treats SYN packets with capability requests as legacy traffic, and its performance under overload will be dominated by long TCP timeouts. This modification also favors TVA slightly. But as we will see, most TVA transfers finish without or with only a few retransmissions. We set the application timeout value to 10 seconds to speed up simulations. That is, we abort a file transfer if it cannot finish within 10 seconds.

The number of legitimate users differs in each sub-topology for each attacker density. We compute the rate of file transfers for each setting such that the file transfers from legitimate users would not congest the bottleneck link. The contention effects we see in the simulations come directly from massed attackers. In each simulation, a legitimate user sends 10 files to the destination.

B. Legacy Packet Floods

The first scenario we consider is that of each attacker flooding the destination with legacy traffic at 1Mb/s. Figure 6 shows the cumulative fraction of file transfer times among all file transfers that are started by legitimate users for TVA, SIFF, pushback, and the current Internet. We see that all TVA transfers complete and the completion time remains small as the attacker density varies from 10% to 80%. The corresponding attack bandwidth varies from 1.25 to 10 times the bottleneck bandwidth. Our design strictly limits the impact of legacy traffic floods, as we treat legacy traffic with lower priority than TVA 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 to suffer losses. 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 equals to the probability a request gets through within 10 seconds. As a SYN packet is retransmitted every second in our simulations, this is equivalent to nine tries, i.e., $(1-p^9)$. When the attacker's density is 80% , p is 90%, giving a completion rate of $(1 - 0.9^9) = 0.61$. This is consistent with the results in Figure 6(d).

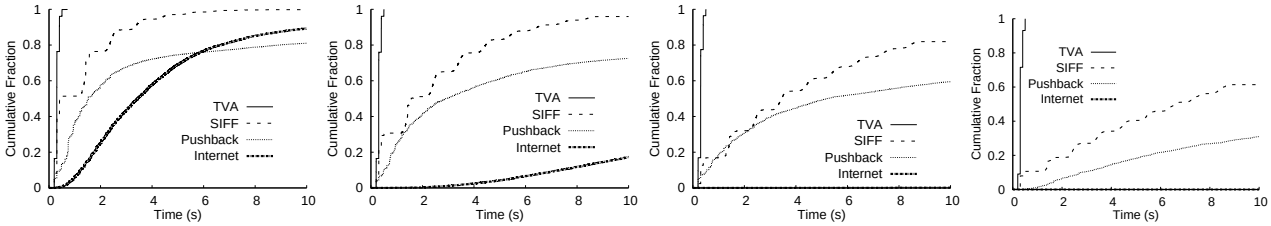
With Pushback, the file transfer time increases as the number of attackers increases, and the fraction of files completed within 10 seconds decreases. This is because the pushback algorithm rate-limits the aggregate traffic from each incoming interface, and it cannot precisely separate attack traffic from legitimate traffic. If legitimate traffic and attack traffic shares the same interface at a bottleneck link, it suffers collateral damage. As the number of attackers increases, more legitimate users suffer collateral damage at multiple hops. Therefore, their file transfer times increase.

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 6: the fraction of completed transfers quickly approaches to zero as the number of attackers increases.

C. Request Packet Floods

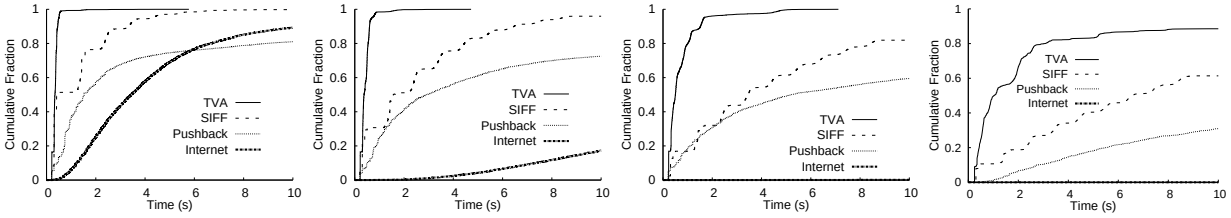
The next scenario we consider is that of each attacker flooding the destination with request packets at 1Mb/s. Attackers that are spoofers send packets with spoofed initial path identifiers. In this attack, we assume the destination was able to distinguish requests from legitimate users and those from attackers.

The results are shown in Figure 7. With TVA, request flooding attacks may cause request channel congestion. In our simulations, all queues at the same level have the same weights. As a result, legitimate users that are far away from the bottleneck link may only have a tiny share of request channel



(a) 10% attackers (b) 20% attackers (c) 40% attackers (d) 80% attackers

Fig. 6. **These figures show the cumulative fraction of the file transfer times among all transfers started by legitimate users. Legacy traffic flooding does not increase the file transfer time of TVA. With SIFF and Pushback, file transfer time increases and the fraction of transfers completed decreases as the number of attackers increases; with the legacy Internet, the transfer time increases, and the fraction of completion approaches zero after the attacker's density exceeds 40%.**



(a) 10% attackers (b) 20% attackers (c) 40% attackers (d) 80% attackers

Fig. 7. **Request packet flooding with spoofed path identifiers may congest TVA's request channel and increases the file transfer time for some legitimate users. When the attacker's density is 80%, more than 80% of legitimate users are isolated from attackers and can complete their transfers in less than three seconds. Less than 14% of legitimate users can not complete their transfers within 10 seconds, as a result of sharing queues with attackers.**

bandwidth, insufficient to send one request packet within one second. Those users will see increased transfer times due to increased queuing delay and request retransmissions. This can be seen from the "tail" part of the TVA transfer time distribution in Figure 7. When the attacker density reaches 80%, spoofed path identifiers cause the bottleneck router to exhaust all its queues. Some legitimate users whose path identifiers overlap with the attackers' may share queues with the attackers. As a result, they suffer collateral damage and cannot complete their file transfers within 10 seconds. Legitimate users that do not share queues with attackers can finish faster than those users in other schemes. In Figure 7(d), more than 80% of TVA transfers can finish within three seconds. This result depends on the topology and the number of queues the bottleneck router can support. In general, fewer queues or more attackers will limit TVA's ability to separate attackers from legitimate users, leading to collateral damage to legitimate users.

TVA's results can be improved if we allocate more bandwidth to the request channel, or assign weights to queues based on the measured request traffic demand when there are no attacks. In the simulations, we strictly rate limit TVA's request packets to 5% of the bottleneck bandwidth. This assumes that the data channel is congested at the same time, and the request channel cannot use spare bandwidth in the data channel. Otherwise, if the bottleneck link is work conserving and the data channel is not congested, request packets may use the available bandwidth in the data channel and encounter less congestion. This will reduce both the queuing delay and the loss rate of legitimate requests.

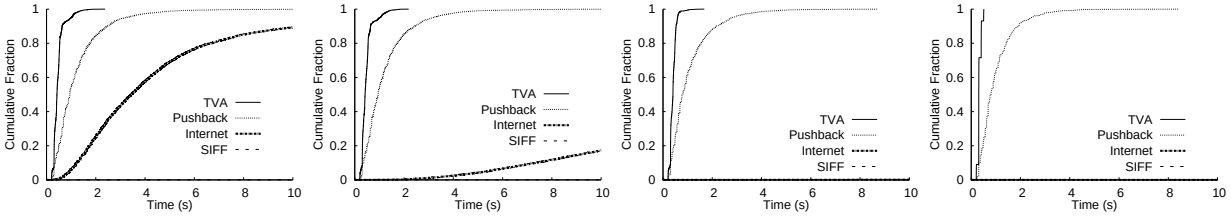
The results for SIFF are similar to those for legacy packet floods, as SIFF treats both requests and legacy traffic as low priority traffic. Both pushback and the legacy Internet treat request traffic as regular data traffic. The results for them are the same as those for the legacy traffic attack.

D. 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 8 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. When the fraction of attackers is less than 80%, a small fraction of transfers take a few retransmissions to finish. This is because there are a large number of users, and after their available bandwidth is reduced by the attack, TCP burstiness causes temporary congestion. But 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 will not be starved.

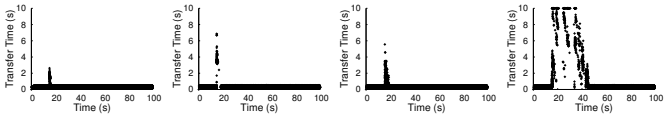
Under the same attack with SIFF, legitimate users are completely starved. Again, this is because the request packets are treated with low priority and are dropped in favor of the authorized attack traffic. We see in Figure 8 that no SIFF transfers complete even when there are only 10% attackers.

Pushback performs reasonably well in this scenario, but its file transfer times still increase. This is because pushback is per-destination based. If legitimate traffic and attack traffic do not share the same destination, legitimate traffic does not suffer collateral damage caused by pushback's rate limiting, but it



(a) 10% attackers (b) 20% attackers (c) 40% attackers (d) 80% attackers

Fig. 8. **With TVA, per-destination queue ensures that the destination and the colluder equally share the access link bandwidth. A few transfer times increase as a result of reduced bandwidth, but all transfers complete.**



(a) TVA high (b) SIFF high (c) TVA low (d) SIFF low

Fig. 9. **X-axis is the simulation time a transfer is started. Y-axis is the time it takes to complete the transfer. 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.**

still suffers congestion losses caused by the attack traffic at the bottleneck.

The legacy Internet treats request traffic and authorized traffic as regular traffic. Thus, the results for the legacy Internet under an authorized traffic attack is similar to those under a legacy traffic attack.

E. 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 case 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 III-C: 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 9 shows how the transfer time changes for TVA with this policy as an attack commences. The attacker density is 80%. There are two attacks: a high intensity one in which all attackers attack simultaneously; and a low intensity one in which the attackers divide into 10 groups that flood one after another, as one group finishes their attack. We see that both attacks last for a short period of time. When the number of attackers increases, the impact of an attack may increase, but the attack will stop as soon as all attackers consume their 32KB capabilities.

Figure 9 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 more pronounced effect on SIFF.

F. Summary

The results presented in this section evaluate the benefits and limitations of the design choices of TVA. The comparison between TVA and Pushback highlights the benefits of capabilities: without capabilities, every data packet may suffer collateral damage; with capabilities, only the first request packet of a connection may suffer collateral damage. The comparison between TVA and SIFF shows the benefits and limitations of treating request packets with the same priority as data packets, protecting the request channel with hierarchical fair queuing, and fine grained capabilities. The higher benefits of TVA come from these additional defense mechanisms.

VI. Implementation

We prototyped TVA using the Linux Click router [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 use AES-based message authentication code to compute pre-capabilities and AES-based Matyas-Meyer-Oseas hash [23] as the second secure hash function to compute capabilities. We use AES because of its superb hardware speed [14]. We implement the path-identifier based hierarchical fair queuing scheme using DRR [29] and HFQ [8].

The purpose of this effort is to check the completeness of our design and to understand the processing costs of capabilities. In our experiment, we set up a router using an AMD Opteron 2.6GHz CPU with 2GB memory. Both the router, packet generator, and packet sink run a Linux 2.6.16.13 kernel. We then use a kernel packet generator to generate different types of packets and send them through the router, modifying the code to force the desired execution path. For each run, our packet generator sends ten million packets of each type to the router. We record the average number of instruction cycles for the router to process each type of packet, averaging the results over three experiments.

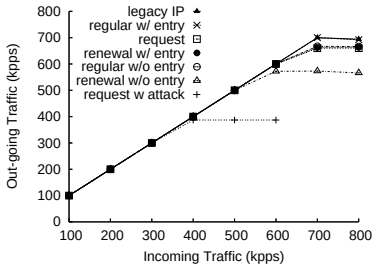


Fig. 10. **The peak output rate of different types of packets.**

Packet type	Processing time	
Request	No attack	Under attack
	313 ns	1378 ns
Regular with a cached entry	9 ns	
Regular without a cached entry	628 ns	
Renewal with a cached entry	341 ns	
Renewal without a cached entry	852 ns	

TABLE I

Processing overhead of different types of packets.

We also evaluate the processing costs of request packets under a request flooding attack. The attack will trigger a router to split hierarchical queues to separate packets with different path identifiers, increasing the processing costs. We set a queue limit of 200K in our experiments, and the maximum depth of a queue to five, because longer paths increase processing costs, and most AS paths are less than five hops long. We use an AS-level topology obtained from the Oregon RouteView server as described in Section V to obtain path identifier distributions. Our packet generator uniformly generates request floods from each path identifier, forcing a request queue to be created for each unique path identifier. We also randomly choose 25% of path identifiers to be spoofable and prepend them with spoofed tags [9]. This forces the router to exhaust all its 200K queues. We then benchmark the cycles to process a request packet, averaging the results over all path identifiers.

Table I shows the results of these experiments, with cycles converted to time. In normal operations, the most common type of packet is a regular packet with an entry at a router. The processing overhead for this type is the lowest at 9 ns. The processing overhead for validating a capability for a packet without a cached entry is about 628 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 852 ns. During request flooding attacks, the processing cost of a request packet increases to 1378 ns.

We also evaluate how rapidly a Linux router could forward capability packets. The results are shown in Figure 10. The output rate increases with the input rate and reaches a peak of 386 to 692Kpps, depending on the type of packet. This compares well with the peak lossless rate for vanilla IP packets of about 694Kpps. All types of packets are minimum size packets with a 40 bytes TCP/IP header plus a minimum capability header of that packet type. Request packet processing under request floods has the lowest throughput, but is sufficient to saturate 5% of a 3Gb/s link.

VII. 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 payload (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. Some router implementations may return more than eight bytes of payload in ICMP messages. In that case, an attacker may obtain pre-capabilities up to that router, but not after the router. If it turns out to be a security risk, a future version of TVA may pad more bytes in request packets, a tradeoff between security and efficiency.

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, and a host sharing a broadcast and unencrypted LAN. 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. A compromised router is just another attacker – it does not gain more leverage than an attacker at the compromised location. So is an attacker that sniffs a sender's capability at a LAN. DoS attacks on a destination will still be limited as long as there are other capability routers between the attacker and the destination. However, senders behind the router or sharing the same LAN with an attacker will be denied service, a problem that can only be solved if senders do not use the compromised router to forwarding packets or by improved local security.

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 compromised 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.

TVA's capabilities cover all connections between two hosts. In the presence of NAT or time-shared hosts, one malicious host or user may prevent all other hosts or users sharing the same IP address to send to a destination. Unfortunately, this problem cannot be easily solved without a better scheme for host or user identity. If future work solves the identity problem, TVA can be modified to return capabilities on a per-host or per-user basis.

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.

VIII. Discussion

A. Deployment issues

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.

B. Limitations

We have constrained our design to modify only the data plane of the network, as modifying control plane may require inter-ISP cooperation and additional control messages and mechanisms to prevent those control messages from being DDoSed. We have also constrained our design to be architectural in the sense that we aim to protect any destination and any bottleneck.

Consequently, designs that relax these restrictions may have different cost and benefit tradeoffs. For instance, designs that aim to protect a bottleneck near a web server [11] may be simpler than TVA, as they can use SYN cookies to prevent source address spoofing, and respond faster in cutting off attack traffic, if filters can be installed faster than attackers consuming their initial capabilities. Similarly, designs [16], [21] that assume the bottleneck link is always close to a destination may also be simpler than TVA.

In addition, if we relax our design space to allow modifications in the control plane, capability-based systems can be made more scalable than TVA. For instance, if a router can send rate-limit messages to an upstream neighbor when a request queue identified by the neighbor's tag is congested, the router may reduce the impact of request flooding with a small number of queues. Presently, without path spoofing, a TVA router may require as many queues as the number of unique path identifiers to separate legitimate users from attackers. With path spoofing, the number of queues required for perfect isolation is topology dependent, and may grow exponentially with the network diameter. A router with a limited number of queues may not be able to protect all legitimate users.

TVA assumes that end systems have effective policies to differentiate attack traffic from legitimate traffic. Effective policies are an area for future study.

C. Capabilities versus Filters

In [5], Argyraki et al. discussed the limitations of network capabilities. Most design challenges faced by a capability-based design are applicable to a filter-based design. For instance, in a capability-based design, a router may fail to protect legitimate traffic when it does not have enough request queues. Similarly, in a filter-based design, a router may also fail to protect legitimate traffic when it runs out of filters.

We see that the key difference between capability-based designs and filter-based designs is the separation of the request channel and the data channel. The request channel does not need to operate at the wire speed. Intuitively, we think a slow channel is easier to protect because it permits heavier protection mechanisms. TVA uses hierarchical fair queuing to protect the request channel in an effort to balance complexity and effectiveness, but other work [32] may use different mechanisms for different tradeoffs. Even in the case that the request channel is not completely protected from attack traffic, collateral damage only slows down the first request packet of a connection. If a

connection involves more than one packet from each end, then a capability-based design can protect the subsequent packets. In contrast, in a filter-based design, if collateral damage exists due to filter shortage, every packet will suffer.

IX. Conclusion

We have motivated the capability approach to limit the effects of network denial-of-service attacks, and presented and evaluated (a revised version of) TVA, the first comprehensive and practical capability-based network architecture. As a complete system, it details the operation of capabilities along with protections for the initial request exchange, consideration of destination policies for authorizing senders, and ways to bound both router computation and state requirements. We evaluate TVA using a combination of simulation, implementation, and analysis. Our simulation results show that, when TVA is used, even substantial floods of legacy traffic, request traffic, and other authorized traffic have 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 show that our design will be able to achieve a peak throughput of 386-692 Kpps for minimum size packets on a software router. 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. We hope that our results will take capability-based network architectures a step closer to reality.

References

[1] Appendix. `http://www.ics.uci.edu/~xwy/publications/tva-appendix.pdf`.

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

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

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

[5] K. Argyraki and D. R. Cheriton. Network capabilities: The good, the bad and the ugly. In *Proc. of ACM HotNets*, 2005.

[6] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker. Off by default! In *Proc. of Hotnets-IV*, 2005.

[7] P. Barford, J. Kline, D. Plonka, and A. Ron. A Signal Analysis of Network Traffic Anomalies. In *Proc. of IMW*, 2002.

[8] J. C. R. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Transactions on Networking*, 5(5):675–689, 1997.

[9] R. Beverly and S. Bauer. The spoofer project: Inferring the extent of source address filtering on the internet. In *Proceedings of USENIX Steps to Reducing Unwanted Traffic on the Internet (SRUTI) Workshop*, pages 53–59, July 2005.

[10] The CAPTCHA project. `http://www.captcha.net/`.

[11] M. Casado, A. Akella, P. Cao, N. Provos, and S. Shenker. Cookies Along Trust-Boundaries (CAT): Accurate and Deployable Flood Protection. In *In Proc. of USENIX SRUTI*, 2006.

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

[13] M. Handley and A. Greenhalgh. Steps Towards a DoS-Resistant Internet Architecture. In *ACM SIGCOMM FDNA Workshop*, 2004.

[14] A. Hodjat, D. Hwang, B.-C. Lai, K. Tiri, and I. Verbauwhede. A 3.84 gbits/s AES crypto coprocessor with modes of operation in a 0.18 m CMOS technology. In *ACM Great Lakes Symposium on VLSI*, 2005.

[15] A. Hussain, J. Heidemann, and C. Papadopolous. A Framework for Classifying Denial of Service Attacks. In *ACM SIGCOMM*, 2003.

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

[17] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-Sale: Surviving DDos Attacks that Mimic Flash Crowds. In *2nd NSDI*, May 2005.

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

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

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

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

[22] P. McKenney. Stochastic fairness queuing. In *Proc. of IEEE INFOCOM*, 1990.

[23] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*, chapter 9. CRC Press, 1997.

[24] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. The Spread of the Sapphire/Slammer Worm. `http://www.cs.berkeley.edu/~nweaver/sapphire/`, Jan. 2003.

[25] D. Moore, C. Shannon, and J. Brown. Code Red: A Case Study on the Spread and Victims of an Internet Worm. In *Proc. of IMW*, 2002.

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

[27] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of HotNets-I*, 2002.

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

[29] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *ACM SIGCOMM*, Aug. 1995.

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

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

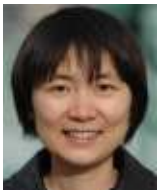
[32] D. Wendlandt, D. G. Andersen, and A. Perrig. FastPass: Providing First-Packet Delivery. Technical report, CMU CYLAB, 2006.

[33] 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.

[34] 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.

[35] X. Yang, D. Wetherall, and T. Anderson. A DoS-Limiting Network Architecture. In *ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[36] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. In *Proc. of ACM SIGCOMM*, Aug. 2002.



Xiaowei Yang (SM '99-M '05) is an assistant professor of Computer Science at the University of California, Irvine. Her research interests include congestion control, quality of service, Internet routing architecture, and network security. She received B.E. in Electronic Engineering from Tsinghua University in 1996, and a Ph.D. in Computer Science from MIT in 2004.

David Wetherall is an Associate Professor of Computer Science and Engineering at the University of Washington and Director of Intel Research Seattle. His research interests are centered on networks, and distributed systems. He received a B.E. in electrical engineering from the University of Western Australia in 1989, and a Ph.D. in computer science from MIT in 1998. Wetherall received an NSF CAREER award in 2002 and became a Sloan Fellow in 2004.

Thomas Anderson is Professor of Computer Science and Engineering at the University of Washington. His research interests concern the practical issues of constructing secure, reliable, and efficient computer and communication systems. He received a A.B. in philosophy from Harvard University in 1983 and a Ph.D. in computer science from the University of Washington in 1991. Anderson is an ACM Fellow and has been awarded a Sloan Research Fellowship, an NSF Presidential Faculty Fellowship, and the ACM SIGOPS Mark Weiser Award.