Deployable Filtering Architectures Against Large Denial-of-Service Attacks

Felipe Huici

Telephone: +44 (0)20 7679 0401

Fax: +44 (0)20 7387 1397

Email: f.huici@cs.ucl.ac.uk

URL: http://www.cs.ucl.ac.uk/staff/f.huici

Supervisor: Prof. Mark Handley

A document submitted in partial fulfillment

of the requirements for a

Doctor of Philosophy

at the

University of London

Department of Computer Science

University College London

December 17, 2009

Abstract

Denial-of-Service attacks continue to grow in size and frequency despite serious under-reporting. While several research solutions have been proposed over the years, they have had important deployment hurdles that have prevented them from seeing any significant level of deployment on the Internet. Commercial solutions exist, but they are costly and generally are not meant to scale to Internet-wide levels.

In this thesis we present three filtering architectures against large Denial-of-Service attacks. Their emphasis is in providing an effective solution against such attacks while using simple mechanisms in order to overcome the deployment hurdles faced by other solutions. While these are well-suited to being implemented in fast routing hardware, in the early stages of deployment this is unlikely to be the case. Because of this, we implemented them on low-cost off-the-shelf hardware and evaluated their performance on a network testbed. The results are very encouraging: this setup allows us to forward traffic on a single PC at rates of millions of packets per second even for minimum-sized packets, while at the same time processing as many as one million filters; this gives us confidence that the architecture as a whole could combat even the large botnets currently being reported. Better yet, we show that this single-PC performance scales well with the number of CPU cores and network interfaces, which is promising for our solutions if we consider the current trend in processor design.

In addition to using simple mechanisms, we discuss how the architectures provide clear incentives for ISPs that adopt them early, both at the destination as well as at the sources of attacks. The hope is that these will be sufficient to achieve some level of initial deployment. The larger goal is to have an architectural solution against large DoS deployed in place before even more harmful attacks take place; this thesis is hopefully a step in that direction.

Contents

Chapter 1

Introduction

From its inception, the Internet was designed to efficiently transfer packets and to be flexible enough so that it could accommodate future applications. Dating back to its early days as the ARPAnet, trust was placed on users not to abuse the network, and so security was not, until recently, of great concern. The Internet has evolved significantly since those early days, and people have become used to depending on it for a large variety of activities, from long distance voice and video communications to online banking, e-government and online shopping. However, this increase in activity, specially regarding economic transactions, has made the Internet attractive to the criminal sector, and so the assumption that users can be trusted no longer holds.

While it would be impractical and unrealistic to completely revamp the current Internet to make it secure against attack, it is possible to design incrementally-deployable solutions so that the Internet can continue to grow unhindered.

1.1 Definitions

In a Denial-of-Service (DoS) attack, an attacker tries to exhaust one or more resources from a victim so that it cannot service its legitimate clients. Resources that can be attacked range from a server's CPU and memory to its network capacity. DoS attacks can consist of exploiting a software bug, causing a server to crash, or flooding it with traffic to overwhelm it.

Since it would be difficult for an attacker to exhaust a server's resources using only his system, he can use worms or other malicious software to take over the systems of other unsuspecting hosts, turning them into "bots" that do his bidding. Attacks involving such bots are called Distributed Denial-of-Service attacks (DDoS) and can be quite harmful depending on the size of the botnet used to perpetrate them. DDoS attacks are particularly effective at performing flooding attacks, in which the victim (or even its network link) becomes saturated by attack traffic.

One variation on the DoS theme is called a *reflection attack*, in which the attacker sends

a packet to a server but spoofs the source by putting the target's address instead of his in the header; the server then replies, but this traffic ends up going to the victim rather than the attacker. This attack is usually coupled with an *amplification attack*, in which the reflection server sends replies that are substantially bigger than the requests that triggered them.

1.2 Problem Description and Motivation

Despite serious under-reporting, the number and size of flooding DoS attacks are growing at an alarming rate. ISPs are under constant pressure to upgrade access speeds, often leaving little time to implement security measures: it is precisely these high-speed yet largely unmonitored links that provide a powerful base from which to launch attacks. This, combined with a continuous stream of exploits, ensures that attacks will keep growing both in number and size. In its biannual report [56], Symantec reported an average of 5,200 attacks per day for the first half of 2007, and these only included TCP SYN request flood attacks. Arbor Network's annual survey of tier-1 and tier-2 security engineers reports that DDoS attacks are reaching new heights of traffic, up to 40 Gb/s in the past year [53].

The character of these attacks is also changing. While in the beginning they were mostly perpetrated by amateurs seeking to boast about their personal achievements, recent years have seen a shift towards more sinister and diverse motives. In the more common DoS attack, a company's links are flooded and a ransom note sent demanding payment in exchange for having the attack stop; not paying the ransom can have drastic consequences [50]. Despite under-reporting (companies are loathe to advertise the fact that they have paid, lest they invite future attacks), these types of attacks are on the rise [41]. Other financially-motivated attacks are perpetrated by businesses trying to eliminate or gain an advantage over their competitors [45]. Further, attacks can be motivated by political reasons [9], ideological reasons [44] and can be even caused by implementation mistakes [59].

Botnet size is also increasing: reports from 2005 indicate that Dutch bot-herders may have controlled as many as 1.5 million hosts [65], while a more recent report states that a gang's botnet included almost two million bots [38]. To make matters worse, shutting these down will become increasingly difficult, since their command-and-control servers are starting to be structured on a peer-to-peer model. Certain security techniques such as Address-space Layout Randomization (ASLR) are aimed at preventing exploits. However, research has already demonstrated that ASLR slows the attack down, but does not prevent it [51]; and of course, there are many older systems connected to the Internet that do not implement these new techniques, and so are vulnerable. In addition to this, attackers will always be able to exploit machines by

using email, P2P and other social engineering techniques [56]. We believe that these factors ensure that botnets will be around for a long time to come.

Designing defenses against large, distributed attacks is intrinsically difficult for several reasons. First, while the core of the network would seem like the perfect location to place defense mechanisms, ISPs there have few if any direct incentives to deploy new measures. Further, change in the core is likely to require hardware change to core routers. Even if reconfiguring routers is all that is needed, operators are generally reluctant to make important modifications without clear incentives. Second, defenses implemented at the client hosts are just as problematic. Unless such defenses were hardware-based, it is likely that they could be broken or exploited by an attacker. Third, source address spoofing complicates matters further, making the task of tracing the sources of malicious traffic difficult. In addition, any DoS defense mechanism that does not take spoofing into account can be subverted by an attacker into denying service to an unsuspecting victim, turning the mechanism into a DoS tool in its own right. Fourth, any proposed solution must be flexible enough to allow the development and deployment of new applications, one of the main reasons behind the Internet's amazing growth. Finally, any practical solution needs to provide clear incentives for initial deployment or it will fail regardless of technical merit. Despite these difficulties, steps must be taken to at least mitigate these attacks so that the Internet can continue to expand.

1.3 Requirements

It is clear that the DoS problem will not disappear but rather continue to grow. It is also clear that to effectively defend against DDoS attacks, malicious traffic must be blocked close to its sources. Although OS security is improving, end-hosts themselves are likely to remain poorly managed and insecure for the foreseeable future, so any solution must be network-based. Unfortunately, we cannot enlist the help of the network core; the cost of any mechanism in core routers is significantly greater than at the edges. Perhaps more importantly, transit ISPs actually suffer very little from DDoS problems, so have very little incentive to invest in new technologies or even perform router reconfigurations. This leads us to the inevitable conclusion that the only place where these problems can be tackled is at the edge ISPs and networks; only here does anyone have both the incentive and the means to tackle the problem.

Although a number of *pro-active* solutions to DDoS have been proposed, none can be deployed solely at edge ISPs with no changes to hosts or to core routers. Thus it appears that any IP-level solution that is actually realistic for deployment must also be *reactive*: essentially detect problem traffic and request that it stop. Such a filtering mechanism must be designed

to prevent misuse. For example, it must not be possible for spoofed traffic to trigger a filter to be installed that blocks legitimate traffic. In addition, the filtering mechanism itself must be robust against DDoS. As far as possible, this implies that its control plane should be relatively decoupled from the data forwarding path. If, for whatever reason, the filtering control plane were to fail, the situation must not be worse than if the DoS solution had not been deployed.

From this discussion arise the following requirements for a solution:

- Must be able to mitigate even large, distributed attacks
- Must be deployable in the current Internet without changes to end-hosts or core routers
- Must not be possible to use it as a DoS tool
- Must have the right incentives for deployment
- Must allow for future innovation
- Must achieve all of these with minimal mechanism

1.4 Thesis Overview

In this thesis we present three different and novel architectures against DDoS attacks that aim to meet the requirements mentioned in the previous section. Chapter 2 covers related work both from the research and commercial fields, and explains why despite several years of efforts we are still not any closer to deploying solutions that meet the requirements discussed above. Chapter 3 presents the three proposed architectures, giving a detailed explanation of how each of them works. Chapter 4 provides a detailed investigation of the baseline performance of the platform used for the implementation of the architectures' elements. Chapter 5 discusses the implementation of the various elements of each of the architectures, presenting performance results obtained using off-the-shelf hadware on a real network testbed. Chapter 6 analyses these results, comparing the architectures and discussing remaining issues. Finally, chapter **??** concludes.

Chapter 2

Related Work

Despite several years of effort both in the research community and in the commercial field, Denial-of-Service attacks continue to grow in frequency, size and effectiveness. This section discusses why the proposed solutions have failed to deal with the problem.

2.1 Research Community

While Denial-of-Service attacks have gained considerable notoriety recently, they have existed since the late 1990s. As a result, research in the field has produced numerous and varied schemes aimed at solving or at least mitigating the effects of these attacks, specially those of large, distributed DoS attacks. The discussion that follows is organized using a taxonomy of complete solutions or components that simplify the problem. While some of the proposed approaches do not cleanly fall into one of these categories, many do, and so this classification provides a useful organizational aid.

2.1.1 Traffic Policing and Filtering

Once a large attack has aggregated at a target, there is little the victim can do but drop the malicious packets. For sufficiently large attacks, the processing power needed to do so may exceed the capabilities of the target. Traffic policing aims to alleviate the burden on the victim, either by having network routers filter or rate-limit attack traffic before it becomes fully aggregated, or by preventing this traffic from entering the network in the first place.

Pushback [35] consists of a local mechanism designed to detect and control aggregates at a single router. The router begins by deciding whether or not it is congested by periodically monitoring each queue's packet drop rate to see if it exceeds a certain threshold (this threshold is policy-specific). An aggregate is defined as a collection of packets from one or more flows that have some property in common, like TCP SYN packets. When serious congestion is detected, the router attempts to identify the responsible aggregates by applying a clustering algorithm. If this algorithm fails to find a well-defined aggregate, no action is taken. Once an aggregate is

identified, its rate is limited so that the combination of its new rate with the rates of all other (well-behaving) flows arriving at the router does not exceed a certain configured value. The congested router will then send a pushback message asking the immediate upstream routers that send the bulk of the traffic for an aggregate to rate-limit it; routers receiving this message can then recursively propagate it upstream. These pushback messages are then sent periodically as long as the attack persists. The downstream router now encounters the problem of deciding when to stop the request for pushback, since it cannot tell whether the high-bandwidth aggregate has stopped because of the upstream filtering or because the attack has ceased. To resolve this, the upstream routers send out feedback messages reporting how much traffic from an aggregate they are still seeing.

Another approach relies on server-centric router throttles [64]. An overloaded server installs appropriate throttling rates at distributed routing points such that, globally, server S exports its full capacity to the network, but no more. S wants to keep its load between two thresholds. If the load is too high S will increase the throttles and if the load is too low it will decrease them. The set of internal routers is denoted by R, and each keeps only a few bytes per victim: the victim's IP address and the throttle value. While this increases linearly with the number of victims, this should not present a scalability problem as DDoS attacks are the exception rather than the norm. The throttles are installed at a subset R(k) of the R internal routers, where k is the number of hops between a router and the server S, essentially forming a perimeter around S. Throttles are also installed at routers that are less than k hops away but directly connected to S. When S's load surpasses the upper threshold, S advertises a throttle increase to R(k), and each router in this set multiplies the current rate by a certain factor. Conversely, when S's load falls below the lower threshold, S advertises a throttle decrease to R(k), and each router in the set adds a factor to the current rate. This multiplicative decrease/additive increase mimics TCP's congestion control mechanism.

In D-WARD [39], routers at edge networks actively monitor TCP, ICMP and UDP flows and perform periodic comparison with normal flow models. The attack response is essentially a modification of TCP's congestion control mechanism. The system will apply fast exponential decrease to the rate limit of non-responsive flows; it will enforce slow recovery of rate-limited flows for a certain period; and it will allow fast recovery once a flow has proved that it is well-behaved.

Argyraki et al. [7] base their approach on the observation that while the resources of a victim's router are limited, the Internet routers have enough combined filtering capability to stop even very large DDoS attacks. Active Internet Traffic Filtering (AITF) assumes that an end

host can identify when it is being attacked and can single out, with high probability, a border router close to the source of the attack traffic. The victim contacts its AITF-enabled gateway which installs a temporary filter on its behalf. The victim's gateway then contacts the attacker's gateway and asks it to filter the malicious flow. The attacker's gateway in turn asks the attacker to cease the flow or risk being disconnected. Should this gateway refuse to collaborate with the victim's gateway (perhaps because it does not have AITF deployed), the latter will either try to find an AITF-enabled gateway on the same path but closer to itself or decide to filter out the traffic itself. The mechanism resorts to this second option in the worst case when no AITF-enabled gateways exist.

Firebreak [21] forces packets to a protected host to travel through special routers called firebreaks that are deployed near sources of traffic; these firebreaks then tunnel the packets to the protected host. To deal with incremental deployment, all firebreaks advertise the complete set of firebreak addresses into the routing fabric using IP anycast. As a result, packets addressed to any firebreak address are routed to a nearby firebreak. A similar approach is dFence [36], which transparently inserts special middleboxes on the path of traffic destined to hosts experiencing DDoS.

Despite their merits, these solutions have their shortcomings. Pushback, for instance, requires a path of deployed routers between a victim and the sources of the attack, an unlikely scenario during initial deployment. Further, the routers determine that a flow causing congestion is malicious, which may not be the case. The approach of server-centric router throttles suffers from a similar initial deployment issue, in that it assumes the availability of a perimeter of deployed routers. D-WARD has the advantage of placing control routers near potential sources of attack, but it is not clear what incentive a *source* ISP would have to deploy such a mechanism, nor is it trivial to determine whether a flow is behaving or not, specially for large, distributed attacks. AITF relies on middle-of-the-network routers (at least during initial deployment) to perform the actual filtering, and depends upon a variant of IP route record to determine where packets came from; as such, it faces severe deployment issues. Firebreak suffers from several shortcomings. First, the firebreaks use IP anycast to advertise all of the addresses of protected targets. Not only is large-scale IP anycast not well understood nor widely deployed, but advertising in this fashion is likely to present scaling problems. Second, the paper points out that, thanks to IP anycast, traffic from clients whose ISPs do not have firebreaks will be redirected to an ISP who does have firebreaks. However, it is quite unclear what incentive this latter ISP has for accepting this traffic nor why it should deploy a firebreak in the first place if there is the possibility of having to deal with another ISP's traffic. Finally, the scheme suggests

stopping traffic from clients connected to legacy routers from going directly to the target (without first traversing a firebreak) by only advertising the route to deployed ISPs. While this will indeed prevent a legacy router from forwarding packets to a protected target, it assumes that there will be no legacy ISP in the path between deployed ISPs and the target ISP. If there were, the route would not be advertised to the legacy ISP and any deployed ISP behind it would be oblivious to the fact that a route exists. This presents a significant problem, especially during initial deployment. The final approach, dFence, requires a significant amount of state to be kept at the middle boxes in order to track connections, opening itself up to state attacks. Further, the scheme does not give deployment incentives to early adopters.

2.1.2 Service Differentiation and Capabilities

A characteristic of hosts on the Internet is that they do not decide which packets to receive. Schemes in this section aim to empower the receiver so that it will be in charge of deciding who is allowed to send it packets. In general, a client requests permission to send packets, and, if the server is not overloaded and decides that the client is legitimate, the server will respond with a capability token. The client then includes this token in all its subsequent packets and routers along the path to the server verify the validity of the token.

In the Stateless Internet Flow Filter (SIFF) approach [62], the process begins by dividing Internet traffic into privileged and unprivileged, where the former is always given priority over the latter. To set up a privileged channel a client begins by sending an unprivileged explorer (EXP) packet. Each router along the path calculates a z-bit long marking from a keyed hash function with four inputs: the IP addresses of the source and destination, that of the router's incoming interface and that of the previous hop's outgoing interface. The router then left-shifts all previous markings in the capability field of the EXP packet and adds its own marking to the least significant bits. Upon receipt of this EXP packet the server has to set up its half of the privileged channel, so it sends its own EXP packet back to the client, this time writing the markings that the routers provided into the optional capability reply field. Once this second packet arrives, the client copies the value of the reply field into the capability field of a privileged data packet, and begins sending data to the server. Each router on the path now verifies that its marking matches that of the low-order bits in the packet's field: if it does not, the packet is dropped; if it does, the marking bits are moved to the right-end of the field and the packet is forwarded. A data packet will only arrive at the server if it passes the checks of all routers along the path.

In [4], sources are required to obtain a token from the destination before being allowed to send packets. Request-to-Send (RTS) servers distributed along the path between the source

and destination aid sources in obtaining such tokens; these are coupled with Verification Points (VP) that enforce the capabilities specified by the tokens. A source contacts its local RTS server which in turn propagates the request all the way to the destination's RTS server, creating state in all the RTS servers in between. The destination then sends the token back to the source in a symmetric path with regards to the RTS chain, and the source includes the given token in all subsequent packets. Finally, the VP ensures that the source respects the given capability.

In [32], legitimate traffic is separated from attack traffic and protected during an attack. This process is done through cryptographic techniques, so that only the first connection packet will suffer a delay, while all the rest will be privileged. When a server is overloaded, each client (malicious or otherwise) is given a quota for high-priority traffic; when this quota is exceeded, the user will be considered an attacker and it will be rate-limited by a perimeter of routers forming a line of defense.

The approaches presented in this section face important deployment issues: they require changes to both the end-hosts so that they will know to request permission to send and to the network so that it will police traffic travelling to a server. Further, these mechanisms do not provide a complete solution, since it is still possible to DoS the capability request channel.

2.1.3 Overlays

Solutions in this category, as their name indicates, consist of deploying a set of special nodes forming an overlay network protecting potential victims. More specifically, overlays aim to distribute the points of access to victims among a set of outer nodes: all packets must pass through one of these nodes before they are allowed into the overlay network. Should a particular flow misbehave, its corresponding access node can limit its rate or drop it. While any one of these nodes can be singled out and attacked, legitimate users can continue receiving service by connecting to another of these nodes.

In i3 [1–3], the authors argue that hosts would be much better placed to effectively respond to overload if they had control over which packets were dropped. For instance, a host running multiple services may give higher priority to some services than others. As a result, hosts, not the network, should be given control to respond to packet floods and overload. The use of the i3 overlay provides a method of identifying hosts without using IP addresses. Sources send packets to a logical *identifier* and receivers express interest in packets by inserting a *trigger* into the network. Packets are then of the form $(id, data)$ and triggers are of the form $(id, addr)$, where *addr* can be either an IP address or an identifier. Triggers, then, relay packets either to a final destination or to other triggers. This added level of indirection allows a host to stop receiving packets from a particular trigger by simply removing it.

In Secure Overlay Services (SOS) [30], a target or destination forming part of a Chord-based overlay [8] notifies other nodes, called *secret servlets*, of its presence, and has a filtering router drop any traffic that does not come from them. These secret servlets, in turn, compute, using several hash functions with the destination's IP address as input, the identity of other overlay nodes called beacons. The servlets then let the beacons know of their existence. When a client wants to send a message to the destination, it must first be authenticated by a Secure Overlay Access Point (SOAP). Upon success, the SOAP will securely forward the message to a beacon, who will in turn pass it to the secret servlet, and from there through the filtering router finally arriving at the destination. The levels of indirection and the self-healing nature of the Chord protocol make this architecture robust against any of its components failing or being compromised.

In COSSACK [22], each egress point has a component called a *watchdog* that monitors its own network and shares information with watchdog peers. Watchdogs have two main functions: detecting the onset of an attack and informing other watchdogs of the attack signature over a multicast channel. Watchdogs at attackers' networks then perform attack-packet filtering.

Centertrack [55] is an overlay that relies on using IP tunnels to re-route interesting packets directly from edge routers to special tracking routers. In this way a tracking router can determine whether a packet is malicious or not, and, if it is, easily follow the tunnel back to the ingress point that the packet took into the overlay.

While overlays have the important advantage of being deployable without affecting the existing network infrastructure, they do not provide a full solution to the DoS problem since they tend to operate above the network layer, assuming the presence of other mechanisms to protect it. Centertrack does operate at the network layer, but uses a central tracking router or small network of tracking routers, providing an obvious target for a DoS attack.

2.1.4 Source Routing

While source routing is not directly intended as a scheme against DoS attacks, its very nature makes it valuable against them. The requirement that sources specify the path a packet should follow to the destination, coupled with the fact that routers along the way verify the path's validity, guarantees that sources cannot spoof their addresses. The availability of the entire path at the destination not only allows easy trace back of the source, but also enables effortless installation of filters close to the attacker.

In NIRA [63], users specify the inter-domain path that a packet should follow, empowering them to select more optimal routes to destinations while fostering competition among ISPs. Another benefit of this architecture is that it forces sources to specify paths to destination that

routers along the way can verify. As a result, source addresses cannot be spoofed, and malicious flows can be easily tracked back to their sources. The general process begins with each host knowing its route to the core of the Internet. When a source S wishes to send to a destination D, it requests that D send it its topology information (D's route to the core). Upon receiving this, S will combine its route to the core with D's route to the core to form a complete path from source to destination.

The *Wide-Area Relay Protocol* (WRAP) [6] is a proposed new protocol running over IP and is a modified version of IP's *Loose Source and Record Route* (LSRR). WRAP involves the use of four header fields, two in the IP header and two in the WRAP header. IP's source address header is modified to always contain the IP address of the source-host or last relay (a WRAP-enabled router) that has forwarded the packet. Likewise, IP's destination header has the IP address of the destination-host or the next relay. WRAP's *reverse path* (RP) field contains a list of IP addresses of relays already traversed while its *forward path* has a list of IP addresses of relays yet to be traversed. End-hosts (or the edge-system) are in charge of computing and monitoring multiple paths to each potential destination in order to include a valid path in the FP field of the WRAP header. Relays then forward the packet according to the given path, as long as it is valid and ISPs along the path have agreements to carry the traffic. With this mechanism in place an end-host can mitigate a DDoS attacks by identifying the paths whose flows are malicious. If the attack is too large, however, the end-host or network will quickly become overwhelmed and will not be able to filter all of the malicious traffic. The authors suggest using a solution such as AITF to propagate filtering requests to upstream routers who can drop these packets closer to their sources.

The clear advantage of source routing in combating DoS attacks is that it provides an enabling mechanism for easy deployment of filters near the attack's source as well as easy trace back to the source, rendering packet marking schemes unnecessary. However, source routing requires wide deployment among routers in order to function. Further, NIRA introduces a new addressing scheme that would require significant changes not only to routers and providers but also to end-hosts. If the numbers of addresses per host is high, NIRA may also prove prohibitive to resource-limited devices. While WRAP indicates that no changes to hosts are needed if a "translating" function is added to edge routers, this scheme also demands changes in routers as well as significant packet remarking upon forwarding.

2.1.5 Proof-of-Work

Denial-of-Service attacks are possible because sources are allowed to send packets even if destinations do not want to receive them. Proof-of-work schemes rely on the destination or server

issuing a computationally-intensive puzzle to its clients; it will then only accept packets from those clients that provide the correct answer to the puzzle. Since, in general, attackers send packets at a much higher rate than legitimate users, they will incur serious computational penalties while legitimate users remain largely unaffected.

To provide coverage across all applications and protocols, client puzzles must be placed at the layer that is common to all of them: the network layer. Feng et al. [19] introduce a new protocol that relies on ICMP source quench messages to issue puzzles and on IP options to reply with solutions. The authors consider four different types of puzzles to use in the protocol. Time-lock puzzles require clients to perform repeated squaring operations. While they provide an exact and fixed amount of computational work, it is expensive for the server to generate them. Hash-reversal puzzles forces clients to reverse a one-way hash given the original random input with n bits erased. The disadvantage of this approach is that the granularity of the puzzles' difficulty is coarse: an n bit puzzle is twice as hard as an n-1 bit puzzle. Multiple hash-reversal puzzles improve this granularity, but issuing an increasing number of puzzles to achieve this becomes prohibitive. Finally, hint-based hash-reversal puzzles provide the result of the hash along with a hint whose accuracy can be varied to regulate the puzzle's difficulty. It is this last type that is used in the implementation of the protocol.

Felten et al. [18] introduce a new method of outsourcing puzzles utilizing a distributed entity called a bastion that can be shared among several servers. Because of its distributed nature and its good scalability, the outsourcing mechanism is robust against DoS attacks. The authors present three methods for puzzle construction, time-lock, hash-reversal and "D-H" (Diffie-Hellman), focusing on this last one since it allows client to solve puzzles off-line. The current implementation is directed at TCP and it is left as future work to see if the techniques are efficient enough to work for lower-layer protocols such as IP.

In [47] clients are allowed to bid for service by computing puzzles with difficulty levels of their own choosing. The server under attack allocates its limited resources to requests carrying the highest priorities. Like the previous approach, this paper describes a TCP implementation, and leaves the possibility of a network-layer implementation as future work.

All of these solutions aim to diminish the imbalance between a server's resources and the combined resources of its clients. The fact that servers can generate and verify puzzles very efficiently while clients need to perform expensive calculations shifts some of the control over to the server. These schemes, however, suffer from several problems. First, if an attack is sufficiently distributed, each source of malicious packets (typically a zombie) will only have to spend a relatively insignificant amount of time solving the puzzle. Worse still, when zombies

calculate solutions to puzzles on behalf of the attacker, it is the legitimate but unwitting owner of the system that incurs the performance degradation resulting from these. Further, puzzles do not take into account the possibility that several orders of magnitude in computing power may exist among a server's clients.

2.1.6 Anti-spoofing

As their name suggests, these schemes have the ultimate aim of distinguishing between genuine packets and those whose source address does not reflect where they actually originated. While they do not solve the DoS problem, they simplify the problem by allowing a victim to know where the attack is coming from. Anti-spoofing solutions are quite varied, differing even in where the filtering takes place; the following is a sample of some of these approaches.

Hop-count filtering [27] makes the observation that most spoofed IP packets do not carry hop-count values that are consistent with the IP address being spoofed. To determine whether or not a packet is spoofed, the hop-count is first computed. The system keeps a table containing a mapping of IP addresses to hop-counts, and if the hop-count of the incoming packet does not match the value found in the table for the IP address of the packet, then the packet is deemed spoofed. To populate this table initially, a node (presumably a server) collects traces of its clients that contain both IP addresses and the corresponding Time-to-Live (TTL) values. The hop-count value cannot be directly obtained from the IP header since no field in it contains this information. It can, however, be inferred from the TTL field: the hop-count is equal to the difference between the initial TTL value at the source and the final TTL value at the destination. The determination of the initial TTL value is based around the observation that while operating systems do not use a common value to initialize this field, the values used by most of them are only a few.

Another approach is called Route-Based Distributed Packet Filtering (DPF) [42] and it relies on placing anti-spoofing routers in the middle of the network (or adding functionality to existing routers). These routers must have knowledge of what IP address ranges are valid, and so the actual implementation of DPF would require at least an augmentation to BGP or the introduction of a new protocol that would disseminate source reachability information rather than destination reachability information.

Mirkovic et al. propose the *Source Address Validity Enforcement Protocol* (SAVE) [33] in which routers periodically exchange SAVE messages. These messages propagate valid source address information from the source location to all destinations, allowing each router along the way to build a table that associates each incoming interface of the router with a set of valid source address blocks. An incoming packet whose incoming interface and source IP address do

not match any of the table entries of a router is dropped.

A more recent approach called Passport [34] relies on efficient, symmetric-key cryptography to allow autonomous systems (AS) along a packet's path to independently verify its identity. The ASes obtain these keys through a Diffie-Hellman key exchange included in routing messages. In addition, the task of inserting the tokens used as packet identifiers can be off-loaded, so there is no need to modify hosts.

Approaches in the network, such as DPF and SAVE, have the advantage of stopping spoofed traffic near the source or before it has had a chance to aggregate, at the cost of difficulty of deployment. DPF requires that the underlying inter-domain routing protocol disseminates source reachability information. As a result, it faces the significant hurdle of either augmenting BGP or creating a separate routing protocol. Neither one of these suggestions seems feasible on the Internet. Further, a DPF node's ability to filter traffic may be hindered by the existence of multiple paths to a source. Also, one would have to evaluate whether the added overhead in keeping source reachability information in addition to destination information is justified by the benefits arising from doing so. Destination-based solutions such as Hop-Count Filtering are easily deployed since the victim has an obvious interest in doing so; unfortunately, the attacker could easily change the hop-count value of malicious packets, throwing off the mechanism. The cryptographic approach of Passport shows promise, though it remains to be seen whether it will see wide deployment.

2.1.7 Packet Marking

While not DoS solutions in themselves, schemes in this section aim to simplify the problem by providing a mechanism that allows a victim to determine where the malicious traffic came from. To achieve this, routers in the network inform the destination about the paths that packets follow before arriving; alternatively, routers can keep information about the packets they forward. Even if the attacker uses source address spoofing, with these mechanisms in place the victim can determine which flows are malicious and either filter them locally or trace them back towards their sources to filter them farther upstream.

In IP Traceback [49], as a packet travels from the source to its destination, each router along the path probabilistically marks it with partial path information. Since attacks are generally comprised of a large number of packets, this method ensures that, given enough packets, the victim will eventually receive a marking from each of the routers, allowing it to reconstruct the path from it to the attacker. Another approach similar to IP Traceback is proposed by Bellovin et al. [11] and is called ICMP Traceback. As packets flow through a router it generates, with low probability, an ICMP message to the destination containing the router's IP address. If a

flow has enough packets, as is the case during an attack, the victim will receive an ICMP packet from each of the routers along the path and it will be able to trace the attack back to its sources.

A slightly different approach is called Hash-based IP Traceback [54]. This system allows tracing a single packet back to its origin, relying on the deployment near routers of components called Data Generation Agents (DGAs) that record information about packets as they are forwarded. Clearly storing complete packets at the DGAs would present an insurmountable storage capacity problem, so this scheme has DGAs use Bloom filters to reduce the amount of space needed to store each packet, with the added benefit of maintaining the privacy of the packet's contents.

Another scheme called Pi [61] deterministically marks packets so that those that follow the same path will share a Path Identifier. The packet marking is formed piecemeal by the routers on the path from the attacker to the victim, overloading the 16-bit IP Identification field to do so. While this means that the path identifier may not be globally unique, this is not a requirement to providing DDoS protection. Once the markings are in place, filtering can be deployed at the victim or preferably a dedicated machine like a firewall placed on the attack path to the victim. It can be simple, dropping all packets with a certain path identified, or more complex, based on configurable thresholds.

The main advantage of packet marking schemes is that they allow victims to trace packets back to their origin even in the presence of source address spoofing. In addition, this information can be derived post-mortem, after the attack has subsided (in the case of the hash-based approach, the victim would have to query the DGA relatively soon after the router forwarded the packet). However, packet marking has two serious obstacles. First, it requires significant deployment in order for the victim to receive markings from all (or most) of the routers along a packet's path. Even Pi, with its less stringent requirements, needs around 50% deployment to guarantee deterministic markings. While the schemes can be implemented using existing equipment, there is little incentive for deployment: the farther upstream a router is from the victim, the less it will be affected by an attack, since distributed attack traffic aggregates at the victim and far away routers are unlikely to be seriously affected by it. Second, packet marking, specially the probabilistic kind, depends upon a large number of packets flowing through a path, an assumption that does not always hold for distributed attacks. The hash-based approach does not have this problem; however, despite the use of efficient Bloom filters, it will eventually encounter storage space limitations since network bandwidth increases faster than memory access speeds.

2.1.8 Others

In PRIMED [58], filters are placed close to hosts and trained so that malicious traffic is dropped or given lower priority than legitimate traffic. However, it assumes that most IP packets are not spoofed; while this may be true in the current Internet, deploying a mechanism like PRIMED would result in attackers quickly resorting to spoofing. Even without this problem, it is not clear how accurate the filters will be, nor does the study present testbed figures to show the performance of the system under stress.

In a rather radical approach [60], the authors state that a characteristic of DoS attacks is that a small number of bad clients can deny service to a far larger number of legitimate clients. Further, they observe that in many cases bad clients are bandwidth-limited, otherwise they would simply send at higher rates to increase the attack's effectiveness. The solution consists of a server under attack asking legitimate clients to send at a higher rate, so that the percentage of the link used up by malicious traffic is decreased. Despite its merits, operators will likely be reluctant to deploy a DoS protection scheme that increases traffic during an attack.

2.2 Commercial Solutions

Commercial DoS solutions exist from companies like Cisco [14], Arbor [5] and Mazu [37]. However, these do not inter-operate and are meant to provide point or site protection against DoS attacks, not scale to architectural levels. In addition, if an attack is large enough it can still bring down a site protected by this type of hardware. Finally, these boxes are expensive, and a large number of ISPs and server owners cannot afford them. An alternative are scrubbing centers [46], in which traffic is sanitized before reaching a victim. While effective against some of the attacks currently taking place, it is not clear whether they would be able to withstand a massive attack. Further, neither this protection nor the others mentioned in this section come cheaply, and so are not affordable to all victims of DDoS attack.

2.3 Conclusions

While many of the research solutions discussed above show promise, they tend to place new requirements on the network or end clients, resulting in difficult initial deployment issues. From the more radical source routing approaches to packet marking and capabilities schemes, a requirement is placed to implement significant changes on the network and sometimes even the end-hosts, rendering these solutions impractical in the current Internet. While overlays do present a more feasible deployment story, they do not provide a full solution to the DoS problem since they tend to operate above the network layer.

Another common problem among research solutions is deployment incentives. Often these are completely overlooked or are misaligned: those parties that must deploy in order for the schemes to work are the ones that see the least benefit from initial deployment. Consequently, the approaches, while technically sound, have little chance of seeing the light of the day on the Internet.

Commercial solutions do not provide a perfect solution either. These rely on buying special boxes or redirecting traffic through scrubbing centers, both expensive options that only larger sites can afford. In addition, it is possible for a very large attack to bring even these defenses down, since the boxes' lack of inter-operability means that their defense cannot be made to scale to architectural levels.

Despite promising work both in the research and commercial fields, there is still no comprehensive, architectural solution to the problem of large, distributed DoS attacks. While several proposals have been put forth, they all suffer from initial deployment and incentives issues. The next chapter presents a series of architectural solutions to the DoS problem that aim to address these issues while requiring only off-the-shelf hardware to be implemented.

Chapter 3

Architectures

The goal is to design and implement an architecture to combat large, distributed Denial-of-Service attacks. In addition, the architecture has to be incrementally deployable and provide clear incentives for early adopters or it will never see the light of day. Indeed, many of the solutions presented in the related work solve the DoS problem to various degrees, but they fail to do so in an incrementally deployable manner, and, as a result, the DoS problem continues to grow. Further, the solution needs to take spoofing into account or it may be turned into a DoS tool in its own right. While currently most attacks do not spoof, this is certain to change if an anti-DoS mechanism is put in place.

The proposed contribution consists of three architectures against DoS and their evaluation. The architectures are incrementally deployable, providing incentives for early adopters while improving in effectiveness as deployment continues. Each of the components of the architectures will be built on different types of cheap, off-the-shelf-hardware and their performance evaluated on a network testbed to see how they cope when dealing with a load similar to the one they would experience while enduring a large attack. The testing will also aim to understand which extreme conditions would make a particular component fail. The ultimate goal and contribution will be to give confidence that the architectures presented would be effective against large DoS attacks on the Internet.

The work in the first architecture stemmed from a paper by Handley and Greenhalgh [24] describing seven steps that could be taken to protect the Internet against DoS attacks. The paper's approach was purposely radical, ignoring deployment issues. The aim of the architecture described in section 3.1 is, consequently, to retain as many of the advantages described in the paper while keeping deployability in mind. The second architecture (section 3.2) makes significant changes to the first one, resulting in important improvements in terms of deployability and making it more likely to be able to implement it using off-the-shelf hardware. The final architecture, described in section 3.3, eliminates a significant amount of complexity from the

previous ones, making it easier to deploy on the Internet.

3.1 Routing and Tunneling Architecture

Ideally we would like to protect all Internet hosts, but realistically it is usually servers that present the biggest target. We propose, consequently, to provide a protected virtual net for those servers that wish to be better defended. An ISP providing such protection could charge a premium fee for the service, giving a clear incentive for deployment. In [55] the author presents a similar approach to the trace back problem that works for small DDoS attacks in the context of a single ISP. Our solution is intended to extend to a network of many collaborating ISPs, but in this section we first examine how our solution might initially be deployed at a single ISP, before examining how multiple ISPs might cooperate.

3.1.1 Single ISP Architecture

The first step is to designate certain subnets of the IP address space as server subnets: these will receive additional protection from attack. We refer to these subnets collectively as the *server-net*; conceptually they are within a protection boundary ringed by control points. Traffic from the public Internet must traverse one of these control points on its way into the server-net.

A condition of being a server-net host is not being permitted to send directly to other server-net hosts. This constraint prevents hosts inside the server-net from attacking other hosts inside the server-net, and prevents server-net hosts being exploited as relays in reflection attacks on other server-net hosts[1]. It also helps slow the spread of worms within the server-net boundary.

The basic functions of a server-net boundary control point are *encapsulation* and *filtering*. At an encapsulator, packets destined for a server are encapsulated IP-in-IP, and sent to a decapsulator located in the server's co-location facility. An ISP must have at least one encapsulator, but maximum benefit will be gained with one encapsulator associated with each Point-of-Presence (PoP) or peering link.

The principal advantage of this architecture is that when a server is attacked, the decapsulator knows precisely which encapsulators the malicious traffic traversed. As a result, it can ask them to filter traffic, stopping the attack some distance upstream of the victim. We note that encapsulation is not the only technique by which this could be achieved. In particular, MPLS tunneling might also be used for this purpose. However IP-in-IP encapsulation has advantages over MPLS. First, the address of the encapsulator can be directly obtained by the decapsulator,

[1] This applies to hosts sending to hosts in other subnets or to server-nets in other domains. It would be harder to prevent communication between hosts in the same subnet since they are directly connected.

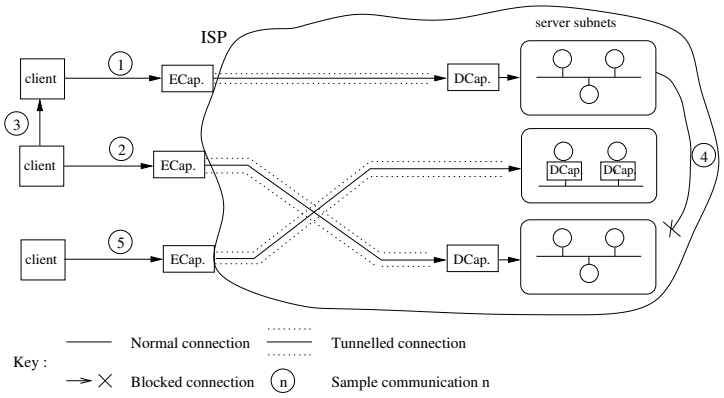


Figure 3.1: *Scenarios for single ISP architecture.*

rather than needing additional mechanisms to reverse map the MPLS labels, but perhaps more importantly it is much harder to extend an MPLS solution inter-domain, which is our eventual goal.

Causing incoming traffic, even traffic originating from within the local ISP, to traverse an encapsulator requires careful control of routing. Routes to the server-net subnets should only be advertised from the encapsulators themselves, to ensure that there is no way to bypass them and send directly to the servers. In addition, the decapsulator addresses should be taken from the ISP's infrastructure address space, which (according to best current practice) should never be advertised outside of the ISP's own network. This prevents an attacker directly flooding a decapsulator associated with a server.

Possible communication paths within this architecture are illustrated in Figure 3.1. Flows 1 and 2 show the typical scenario with packets from a client passing through an encapsulator, being sent to a decapsulator, and finally arriving at the servers. Flow 3 is client to client and is unaffected. Flow 4, from one protected server to another, is disallowed to prevent reflection attacks and the spread of worms. Finally, flow 5 shows a protected server choosing to perform decapsulation itself.

Server→client traffic could be sent via the reverse of the incoming tunnel, or it could be forwarded natively. Either reverse path for the traffic is feasible with the proposed architecture, it should be an operational decision as to which is used. Tunneling has the benefit that a smart encapsulator can view both directions of a flow, allowing it to monitor and filter traffic more intelligently. However, this requires forwarding state at the decapsulator that is set up based on observed incoming traffic, and this state might be vulnerable to DoS if source address spoofing is used.

One solution is for the server to switch dynamically from a native to a tunneled reverse path

once a connection is fully established and is therefore known not to be spoofed. Traffic with a tunneled reverse path can then be forwarded from encapsulator to decapsulator with higher diffserv priority, which might lessen the effect of flooding attacks that spoof source addresses.

The goal of a server-net when deployed at a single ISP is to allow automated filtering of unwanted traffic at the ingress point of that ISP. To perform such automated filtering requires a detection infrastructure in place, located so that it can monitor traffic to the server. Possible locations for this would be in the decapsulator, in the server, or on the path between the two.

Once a flow has been identified as hostile, the decapsulator needs to be informed, and it in turn informs the encapsulator, which installs the appropriate filter. The use of infrastructure addresses between decapsulator and encapsulator is the first line of defense against subversion of the filtering capabilities, as it should simply not be possible for normal Internet hosts to send filtering requests directly to an encapsulator. Behind this first line of defense, the signalling channel between the decapsulator and the encapsulator should be secured. Simple nonce exchange may be sufficient to protect against off-path attacks, given that a compromised router on the path can already cause DoS. Public-key-based solutions are, of course, also possible.

The benefits are clearly greatest for large ISPs hosting server farms and running a large number of encapsulators. Such ISPs will typically have many peering points with other large ISPs, and these peering points will be both geographically and topologically distributed. Thus traffic from a large distributed attack will be spread across many encapsulators because it will be entering the network from many neighboring ISPs, and so it will be stopped closer to its origin, before it has aggregated to the point where it can cause serious damage.

Smaller ISPs still benefit because their customers can control their degree of exposure, but a large enough attack is likely to overwhelm all incoming links. Nothing an ISP can do *by itself* will help in such circumstances.

3.1.2 Inter-ISP Communication

The benefits of a server-net increase as ISPs co-operate. Extending the protection boundary of the server-net to include the server-nets of co-operating ISPs moves the control points nearer to the sources of the DoS traffic. As we extend the protection boundary, distributed attacks become less concentrated at any particular control point, since traffic from each attacking host enters the server-net through its local encapsulator. Traffic that would previously have traversed the peering link uncontrolled now traverses the peering link encapsulated, within the server-net control boundary.

The general idea is that traffic enters the server-net at the server-net ISP closest to the traffic source, and is then tunneled from that ISP's encapsulator directly to a decapsulator at the

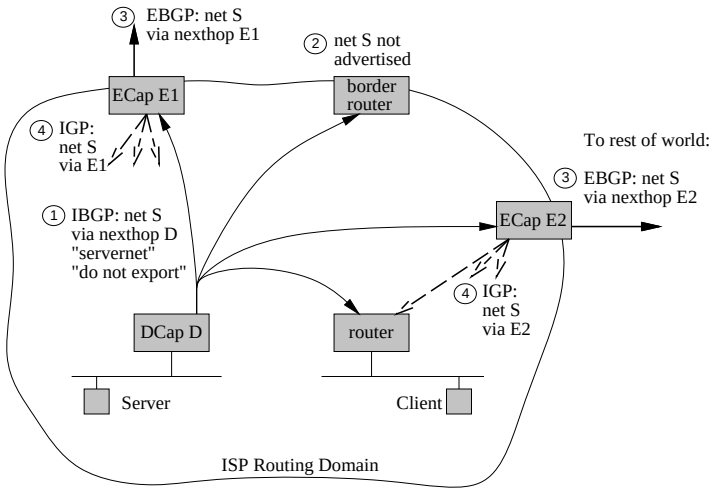


Figure 3.2: *Route propagation for single ISP server-net*

destination ISP border. At the destination ISP, the traffic is decapsulated and re-encapsulated to get it to the final decapsulator near to the server. In principle it would be possible to tunnel direct from the remote ISP to the server subnet, but this assumes a degree of trust between ISPs that seems unlikely and unnecessary.

Traffic originating within an immediate neighbor ISP is forwarded natively to the border of the destination ISP where it is encapsulated as in the single ISP case.

3.1.3 Single ISP Routing

The basic requirements of routing to implement a server-net within a single ISP are:

- Server-net addresses must only be advertised to the rest of the Internet from the encapsulators. [*operational requirement*]
- The encapsulators need to learn which subnets are in the server-net space, and which decapsulator is associated with each. [*operational requirement*]
- The addresses used by the encapsulator and decapsulator must not be advertised to the outside world. [*security requirement*]
- A server-net subnet must not be advertised to server-net hosts on other server-net subnets. [*security requirement*]

We believe these requirements can be satisfied by current BGP[48] implementations on current router hardware.

The server-net could be manually configured, but in a large ISP this will probably be unfeasible. There are many ways to do this dynamically, but one possible solution is illustrated

in Figure 3.2 and elaborated below.

(i) The decapsulator associated with a server-net subnet advertises that subnet into I-BGP[2]. It advertises its decapsulation address as the BGP next-hop router address. The route is tagged with a special BGP *server-net community*[3]. The route is also tagged with the "*do not export*" community that the ISP normally uses to indicate internal infrastructure routes that should not be exported to peers.

(ii) All border routers in the domain are already configured to not propagate routes to their external peers that have been tagged with the *do not export* community, so server-net routes will not leak to the outside world.

(iii) All encapsulators in the domain receive the server-net routes, and match on the server-net community. They then remove the *do not export* community and the *server-net community* from matching routes, and re-advertise them to external peers with the next-hop rewritten to be their own address. This will draw external traffic to the encapsulators.

(iv) The encapsulators also re-advertise any routes tagged with the *server-net community* into their IGP routing[4]. IGP routes are generally preferred over I-BGP routes on the basis of *administrative distance*[15], so this will cause traffic from clients within the ISP's domain to be drawn to the encapsulators rather than directly to the decapsulators.

(v) The decapsulator routers also receive routes containing the *server-net community* that have been advertised by other decapsulators. The decapsulator installs a black-hole route for these subnets to prevent server-net to server-net communication.

Using routing in this way should satisfy our requirements. It is likely that slightly simpler solutions are possible if we can add BGP Path Attributes, but this is probably best done in the light of deployment experience.

3.1.4 Multiple ISP Routing

To protect its own server subnets, each ISP joining a multi-ISP server-net first runs the mechanisms described for a single-ISP server-net. To peer *within* a multi-ISP server-net, the following changes are needed:

[2]I-BGP: Internal BGP - using BGP to carry routing information between routers within a domain.

[3]A *community* is a locally defined BGP routing tag. The actual value of community to use can be locally decided by the ISP.

[4]IGP: Interior Gateway Protocol - the intra-domain routing within an ISP, typically OSPF or IS-IS.

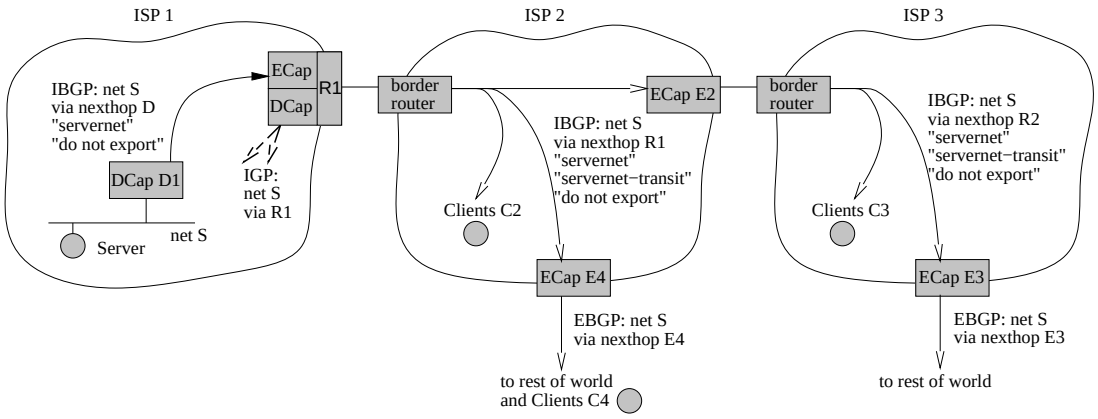


Figure 3.3: *Route propagation for multi-ISP server-net*

(i) Instead of an encapsulator at the border of the ISP, as shown in figure 3.2, a router with both encapsulation and decapsulation capability is used.

(ii) The encap/decap router does not remove the *server-net* and *do not export* communities when transmitting the route to a cooperating neighboring ISP[5].

If the number of server subnets in the multi-ISP server-net were small, then the neighboring ISP can do exactly as described in the single ISP solution. However, in a large server-net, the number of server subnets is likely to exceed the number of routes that can be safely redistributed into the IGP. In addition, even cooperating ISPs are likely to be wary of providing another ISP with a way to inject routes into their IGP. Thus we need to modify the mechanism a little, as shown in figure 3.3.

On receipt at the neighboring ISP, the border router will match on the *server-net* community, and add an additional *server-net-transit* community to these routes, distinguishing them from locally originated server-net routes. Routes with this additional community will *not* be redistributed into the IGP routing by encapsulators.

The result is that traffic from clients within the neighboring ISP's network (C2 in Figure 3.3) will reach the first encapsulator in the destination ISP using native forwarding. On the other hand, traffic from clients that would transit the neighboring ISP (C4 in Figure 3.3) will be encapsulated by the neighboring ISP's encapsulator and tunneled to the decapsulator at the destination ISP, before being immediately re-encapsulated and sent on to the destination server subnet's decapsulator.

[5]This assumes that both ISPs use the same community values to indicate the server-net and do-not-export. If not, it will have to translate to the neighboring ISPs values.

Where more than two ISPs peer within a server-net, routes distributed onward to other server-net ISPs are sent with the BGP next-hop unchanged. Thus traffic will only ever be encapsulated and decapsulated twice:

- Encapsulated at the first encapsulator in the nearest server-net domain to the client.
- Decapsulated and immediately re-encapsulated at the first encapsulator in the destination ISP.
- Decapsulated at the destination subnet.

There is one issue remaining with regards to figure 3.3. There is no guarantee that traffic from clients C3 will traverse encapsulator E2 on its way to R1, and hence be encapsulated. To ensure this does happen requires controlling the inter-domain distribution of routes for R1. In fact the requirement is that routes for such decapsulators only transit between ISPs at the same peerings that server-net routes transit. A very similar use of an additional community tag can be made to preserve this congruence. The principal difference is that such routes are never propagated outside the server-net boundary.

3.1.5 Encapsulation and Filtering

IP-in-IP encapsulation is a standard feature on most routers. Juniper Networks ship a hardware tunnel interface module[28] capable of encapsulation at 10Gb/s that supports 8,000 tunnel virtual interfaces; other vendors no doubt have similar products. Thus current hardware is capable of performing fast tunneling to enough destinations to satisfy even large server-nets.

Most backbone routers also support packet filtering capability. It seems likely that they support sufficient filter rules to cope with attacks on the scales currently seen. In a multi-ISP server-net, any one attack is spread across many encapsulators, making it even harder for an attacker to saturate the filtering capability.

No standard exists for automated pushback of filters, but one would likely emerge if server-nets were widely deployed. In the meantime, a signalling channel would have to work around what is currently available.

Bro[43] can enable filter rules via the command line interface of Cisco routers. This is clunky, but works. In a similar manner, a server-net could use buddy-hosts co-located with each encapsulator. A buddy host would validate that a filter request came from the correct server-net decapsulator address by checking the BGP routing table, and then performing a handshake to prevent spoofing. It would install the filter rule in its local encapsulator using the command line interface. In the long run, we expect routers would directly support such a signalling channel.

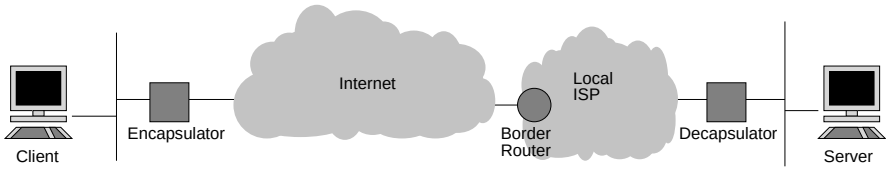


Figure 3.4: *Encapsulation architecture.*

The minimum filter granularity would likely be `<src IP, dst prefix>` to prevent the targeting of other hosts on a victim's subnet. Also possible is `<src prefix, dst prefix>` to avoid an attacker spoofing multiple hosts on the same source subnet.

3.2 Edge-to-Edge Filtering Architecture

This architecture allows hosts under attack to request that the network stop traffic from specified sources before it can aggregate significantly. Similar to the previous approach, three basic mechanisms are required: *marking*, so that the network can know where malicious packets are coming from; *filtering*, so that undesired packets are dropped; and *routing*, so that traffic travelling towards a destination is forced to traverse this filtering.

In this section we introduce an edge-to-edge architecture that implements the three mechanisms mentioned earlier, but does so in a very different way from the previous architecture. The most important change has to do with the location of the control points that mark and filter packets: whereas previously they were placed at the edges of the ISP hosting the server being defended, these are now deployed as close to the clients of the server as possible.

This change has significant benefits. First, it means that these control points never have to handle large traffic aggregates and so can be built and deployed inexpensively. Second, their placement at the edges of the network makes it difficult for an attacker to indirectly DoS a server by attacking the encapsulator. Finally, no filtering or marking is required from the middle of the network, simplifying the deployment story.

The architecture presented in this section also improves over the previous one by distributing *path-agnostic* routes using a separate and extremely robust peer-to-peer protocol, relieving any burden from BGP and removing any disaggregation issues. Finally, it has a simpler initial deployment story, and provides better incentives for it.

But with these advantages come one major problem: it is now possible for some attackers at legacy ISPs to spoof the encapsulation. A key contribution of this architecture is to show that this is not a show-stopper: some additional mechanism is required to be robust, but the

complexity is not excessive. What follows is an explanation of the full solution and these issues in greater detail.

3.2.1 Marking and Filtering

As source IP addresses can be spoofed, if we want to shut down malicious traffic close to its source, a marking mechanism is needed so packets can be traced back to their origin. While many solutions have been proposed for this, a simple mechanism already exists that is just right for the job: IP-in-IP tunneling. The idea is simple: encapsulate all packets near their sources and decapsulate them near their destinations, using the encapsulation header to record the origin network. Two additional boxes are needed for this: an "encapsulator" near the source and a "decapsulator" near the destination; these boxes will also collaborate to filter unwanted traffic. In a later chapter we will show that they can be implemented using off-the-shelf hardware.

During normal operation (see Figure 3.4), a packet from a client will reach a local encapsulator on the path to the Internet. The encapsulator looks up the IP address of the decapsulator, IP-in-IP encapsulates it, and sends the packet to the decapsulator. The source address in the encapsulation header serves to tell the decapsulator which encapsulator forwarded the packet. At the decapsulator, the outer encapsulation header is removed, and the packet is forwarded on to the server.

When a protected server comes under attack, it will send a request to its local decapsulator to filter traffic it deems unwanted[6]. On receipt of a filtering request for a particular source, the decapsulator will wait for the next packet from that source and note down which encapsulator it came through. While this requires the decapsulator to keep state and monitor the traffic going through it, this state is only temporary and will last only until the filtering request is sent.

Once the decapsulator figures out which encapsulator to talk to, it will send out the actual filtering request. Finally, the encapsulator will install the filter, blocking the unwanted traffic. In this way, the server under attack can ask the network to stop sending it undesired traffic, an impossibility in the current Internet.

Of course the full story is not quite as simple as sketched out above, and we will now look at what would be required for this general idea to be viable.

3.2.2 Routing

To perform edge-to-edge encapsulation, the encapsulator needs to know how to map the destination IP address from a data packet to the address of the relevant decapsulator. Essentially this is a routing problem, and this information could in principle be conveyed by BGP. However,

[6]The detection mechanism is beyond the scope of this architecture, but a commercial detection box may be able to perform this role.

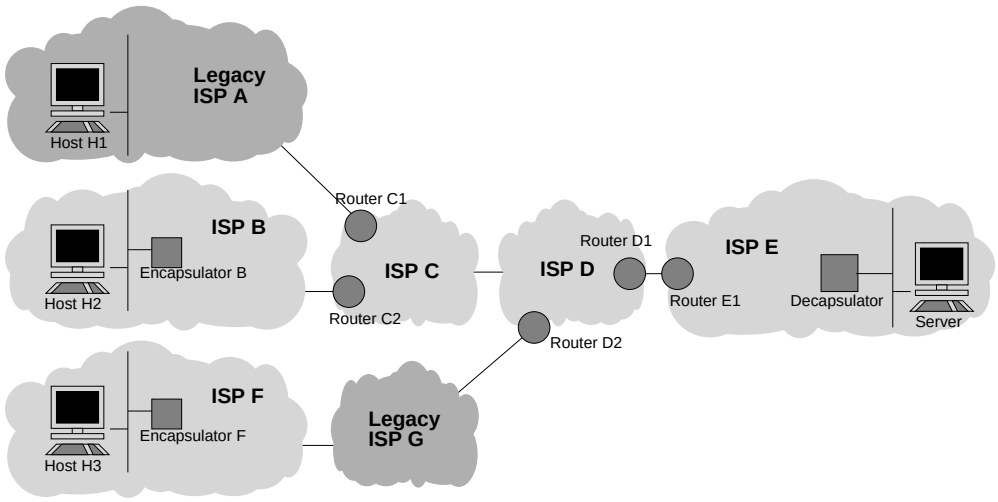


Figure 3.5: *Partial deployment scenarios.*

using BGP would be far from ideal, as the routers in the core of the Internet do not need to know this information. Indeed, the encapsulator does not care about the precise path, but only which decapsulator to tunnel the packet to. Thus, we would be burdening BGP with a great deal of additional information for no good reason. The mapping from network prefix to decapsulator address or addresses is relatively static, so we suggest that a separate dissemination channel be used to distribute these mappings.

As this information is not path-specific, any simple flooding algorithm can be used for this distribution, so long as the data itself is secured. Since no policy is involved, decapsulation routes can be flooded to all of an ISP's neighbors, making this distribution much more robust than conventional inter-domain routing. We will discuss possible routing designs in section 3.2.8.

Making an encapsulator check a large number of digital signatures on startup might be problematic. In reality though, the number of *signatures* required is not related to the number of decapsulators, but rather to the number of origin Autonomous Systems in the BGP routing tables. Each AS can sign as a group all the prefix-to-decapsulator bindings for routes that it advertised, reducing the number of signatures to around 20,000 or so in the current Internet. In addition, there is no need for the edge encapsulators to directly check the signatures themselves. Instead a hierarchy within an ISP can be established, whereby one or more servers receive the routes from neighboring ISPs, check the signatures and only then pass them on to the encapsulators. Thus we believe that signed mappings from destination address prefixes to decapsulator addresses are technically viable and economically feasible, even in the extreme case of a different decapsulator for every /24 subnet in the Internet.

3.2.3 Legacy ISPs

If the mechanism deployed above were ubiquitous, then source-address spoofing by end-systems would be ineffective, and all unwanted flooding attacks could be stopped close to their sources. However, such a scheme must also work with partial deployment, which leaves open the possibility of DoS attacks by end-systems at legacy ISPs that do not deploy encapsulators.

To protect against attacks from such hosts, an ISP protecting a server can involve the border routers at its ingress points. One possible solution would be to deploy a diffserv classifier that prioritizes IP-in-IP traffic destined for the local decapsulators, reducing the effectiveness of bandwidth flooding attacks that attempt to saturate links. Attacks using unencapsulated traffic will then have minimal effect on traffic from networks deploying encapsulators.

Of course the astute attacker will then simply perform encapsulation directly from his attacking end-systems, so that his traffic is prioritized too. By spoofing the encapsulation header, such an attacker at a legacy ISP can make his traffic appear to be coming from many encapsulators. One way to avoid spoofed encapsulation would be to restrict the distribution of routes for the decapsulator addresses, so that the decapsulators are simply unreachable from legacy ISPs. If all the ISPs deploying encapsulators formed a connected graph under normal BGP routing, then this would effectively prevent such attacks from succeeding. However, it is unlikely that such a graph would be connected, at least early in the deployment process, so this solution is not ideal.

Another option would be for all ISPs deploying our scheme to also run an encapsulator for all traffic arriving from legacy neighbors; this is close to the solution described in the previous section. However, performing such encapsulation and filtering on high-speed peering links would be more costly than performing it at edge-links since it could no longer be done with off-the-shelf PCs; this would present an unnecessary deployment hurdle.

Instead, our preferred solution would be to use a single bit in the packets to indicate that traffic destined to a protected server has traversed a legacy ISP. Such a bit is inspired by Bellovin's "evil bit" [10]. ISPs deploying our scheme would install a simple filter at all border routers connecting to legacy ISPs, setting the evil bit in packets arriving from these neighbors. If, along the path, a packet traverses any ISP that does not perform encapsulation, then it is classed as potentially evil. ISPs hosting servers can then choose to prioritize traffic that has come via a path where all the ISPs perform encapsulation.

Figure 3.5 illustrates this mechanism. ISPs A and G are legacy ISPs while the other ISPs have deployed encapsulation. Routers C1, C2, D1, D2, and E1 are conventional border routers configured with simple packet classifiers.

A spoofed encapsulated packet from host H1 destined for the server will traverse ISP A unchanged. Through contractual agreements ISP C will know whether ISP A is a legacy ISP or not; since it is, when the packet reaches ISP C, Router C1 will set the packet's "evil bit", since it knows that the packet came from a legacy neighbor. Router E1 later uses the evil bit to put the packet in a lower priority diffserv class.

A similar packet from host H2 would reach E1 with its evil bit *cleared*, since ISPs B, C and D have all deployed encapsulation; as a result, it is classified as higher priority. However, should the server deem packets from this source to be malicious, it can request that the encapsulator drop them. Even if the host is spoofing, this mechanism will be able to filter the traffic as far as the encapsulator, at which point the problem becomes a local one.

Finally, an encapsulated packet from host H3 would have its evil bit set by ISP D, since although ISP F has deployed encapsulation, provider G has not. Thus all packets arriving at router D2 get the evil bit set, and so receive lower priority at E1. While this may at first seem harsh, it provides the right incentive: customer ISP F will put pressure on ISP G to deploy the scheme (or even switch upstream providers) so that their clients will not be placed in a lower priority queue.

3.2.4 Preventing Abuse of Defenses

Providing a mechanism whereby the recipient of traffic can request that traffic cease is an effective way to defend against all but the most subtle flooding attacks. However, care must be taken to avoid an attacker using our mechanism to deny service to legitimate traffic. We divide the spectrum of possible attacks into two independent vectors:

- Direct attacks that send filtering requests.
- Indirect attacks that generate spoofed traffic with the aim of triggering the defense mechanism to take inappropriate action.

For both cases it is possible to exhaustively enumerate the options open to an attacker, based on what a bot can spoof under differing circumstances and on where the bot is located relative to the systems under attack. These are shown in figures 3.6 and 3.7.

For direct attacks the attacker's options are fairly limited, and we only need to consider spoofed decapsulators in various locations. For indirect attacks there are more cases to consider:

- A bot at a legacy ISP may be able to spoof the encapsulator header, or the client address (so long as a full TCP connection is not needed for the attack) or both.

Spoofed C	Spoofed E	A same E as C	Comments
×	×	×	No spoofing
×	×	√	No spoofing
×	√	×	See figure 3.8:1
×	√	√	Impossible
√	×	×	See figure 3.8:2
√	×	√	See figure 3.8:3
√	√	×	See figure 3.8:4
√	√	√	Impossible

Figure 3.6: *Table of indirect attacks. The letter A stands for the attacker, C the client, E the encapsulator and D the decapsulator.*

Spoofed Request	A on $E \leftrightarrow D$ Path	Comments
×	×	No attack
×	√	A can drop request
√	×	See figure 3.8:5
√	√	A can drop request

Figure 3.7: *Table of direct attacks.*

- A bot at an encapsulating ISP may still be able to spoof the client address if ingress filtering is not performed.
- A bot may be located at the same ISP as the legitimate client he wishes to deny service to, or at a different ISP.

For indirect attacks, spoofing the decapsulator provides no advantage, so we do not consider this case.

We will now examine each of the indirect attack scenarios in Figure 3.6. The first two entries do not constitute an attack and are there merely for completeness.

The third entry (figure 3.8.1) consists of an attacker C located at a legacy ISP who spoofs encapsulator E, but uses his own address C as the client address. The reason for doing this might be that he needs a full TCP connection to trigger a response from the server's defenses. As a result of the spoofing, if the server determines the traffic of C to be malicious, the decapsulator D will ask E to install a filter. This attack is rather harmless unless the attacker possesses a very large number of bots. In such cases it could represent a state-holding attack on E. However, so

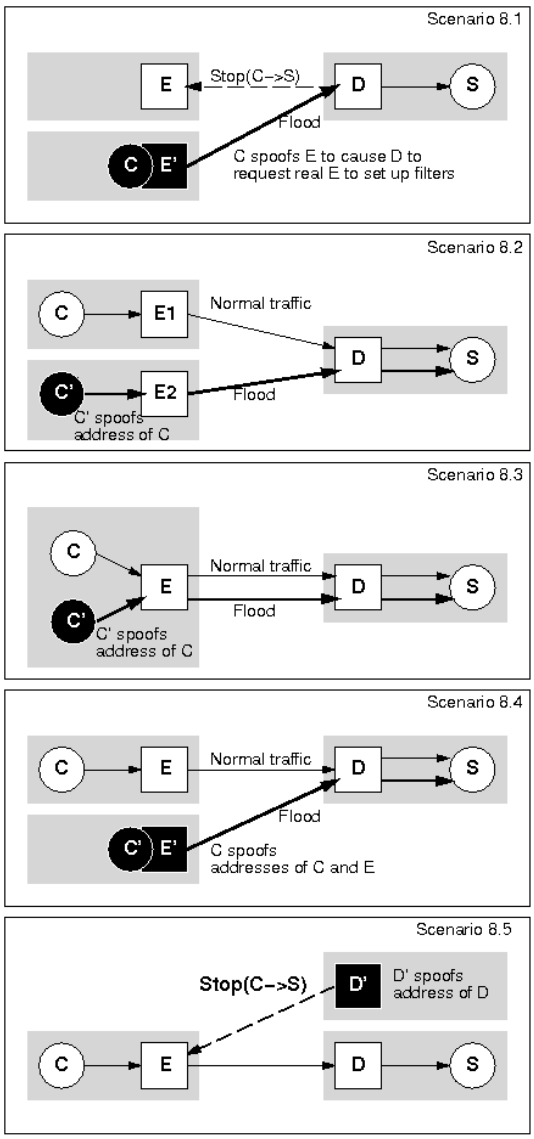


Figure 3.8: *Potential abuse scenarios*

long as E knows which subnets are legitimate client addresses, this attack will not succeed, as E can simply decline to install the filters.

The fourth entry in the table represents an impossible situation, since the attacker would not be able to spoof an encapsulator from a non-legacy ISP.

In the fifth entry (figure 3.8.2) attacker C' sits behind legitimate encapsulator E2 and spoofs traffic from client C, which sits behind encapsulator E1. When D receives a request to block traffic "from C to S", it will wait for the next packet from C to S to arrive and note which encapsulator it came from. However, it is possible that this packet will come from the legitimate C through E1, while the packets that caused the detection mechanism to request the filter may have come through E2; if we asked E1 to install the filter, the attacker would succeed in denying

service to C. In essence, D does not know whether to send the filtering request to E1 or E2. The simplest solution would be for E2 to block this spoofed traffic, as C is not a customer address for E2's ISP. However, this is not always possible, so we would like our solution to work even in the absence of ingress filtering. If every ISP deploying an encapsulator also deploys a decapsulator (a likely scenario) then, using the signed decapsulator routing tables, D would contact C's decapsulator requesting a list of C's encapsulators. As E2 is not on this list, D can safely request that E2 block all traffic from C. The same effect could also be achieved by disseminating the encapsulator list in the decapsulation routing table.

In the sixth entry in the table (figure 3.8.3), both C and C' sit behind encapsulator E. Clearly, neither D nor S can distinguish between traffic from C and C'. S will request E to filter traffic from C (spoofed or not), so C' is successful in denying service to C, although he cannot deny service to other clients of S. In effect the problem has become one that is local to C's ISP, and there are a range of existing solutions available to tackle this, including enabling ingress filtering.

The seventh entry (figure 3.8.4) describes the most subtle attack to defend against. An attacker C' at a legacy ISP spoofs a traffic flood so that it seems to come from C, encapsulated by E. In this way, it could cause S to request a filter at E that would block legitimate traffic from C. If the packets claiming to come from C arrived with the evil bit cleared, we would know that the outer header is valid, a fact that can be propagated to S and the detection system it uses. As a result, S could distinguish between packets arriving on the real path $C \rightarrow E \rightarrow D$ and the spoofed path $C' \rightarrow E' \rightarrow D$ by observing the evil bit. S can now determine whether the unspoofed traffic via E is hostile (in which case it should request a filter) or only the traffic spoofing E is hostile (in which case it should take no action, and rely on prioritization to limit the effects of the attack). What if the path $C \rightarrow E \rightarrow D$ contained legacy ISPs, so that the evil bit is set on all packets? In this case, traffic from both paths would be indistinguishable. To remedy this, upon receiving the filtering request from S, D can provide E with a random number to use to mark subsequent packets (the encapsulation header can contain this). Now D can once again distinguish between the two paths, and all it needs do is to propagate this distinction downstream to S. D can use a diffserv code point to do this, so S's detector can again take the right action to ensure that no legitimate traffic from C is blocked.

The final entry in figure 3.6 presents an impossible scenario, since the attacker cannot spoof an encapsulator from a non-legacy ISP.

We will now proceed to discussing the direct attack cases described in figure 3.7. The first entry does not represent an attack. In the second and fourth entries the attacker sits on the path

between the encapsulator and the decapsulator. These cases are hard to defend against, since the attacker can blackhole legitimate filtering requests. In the fourth case, unless requests are required to be digitally signed by the decapsulator, the attacker can also spoof them to shut down legitimate flows. Such digital signatures should be a mechanism of last resort, reserved only for the case where an on-path attacker is suspected, as they would greatly increase the CPU load on both the encapsulator and decapsulator. However, the encapsulator does already have the relevant key chain to validate such requests, as this is needed to validate prefix-to-decapsulator bindings. In reality though, this is a case we are not greatly concerned about - on path attackers should be rare (the normal bot infection techniques do not apply to routers) and in any event a compromised on-path router has so many other ways to deny service, including simply dropping the packets, that abuse of our mechanism is not likely to make the problem worse.

The third and only remaining entry presents an attack where attacker D' spoofs the address of decapsulator D and requests that encapsulator E install a filter blocking legitimate traffic from client C to server S (figure 3.8.5). This is trivially solved without digital signatures by requiring a three-way handshake, whereby a nonce sent from E to D must be echoed back to E before a filter request will be honored. In this way, even though D' can send a malicious filtering request, it will not be able to respond to E's nonce, since it is not on the path between E and D and will therefore not see it.

One final attack that does not rely on spoofing nor abusing the filtering request consists of flooding the link between the destination's ISP and that ISP's upstream provider. Since the prioritization of packets happens only at the destination's edge router, it may be possible to attack servers by flooding the link. Many ISPs may be able to prevent this by moving the place where diffserv categorization and prioritization occurs from E1 (in figure 3.5) to the upstream provider's edge router D1. As this categorization is static, it requires no active intervention on the part of the upstream ISP, but it does require their cooperation in order to enable it.

3.2.5 When to Encapsulate?

If packets from a client to a server are encapsulated, should the reverse path traffic also be encapsulated? If it is encapsulated, should the forward-path encapsulator serve as the reverse-path decapsulator?

The architecture does not require either, but there are advantages if both are true. If traffic is always bidirectional through the encapsulator, it can pro-actively limit malicious traffic, as described in [31]. Short of mandating symmetry, which seems excessively inflexible, we can still gain these benefits if the encapsulator knows which decapsulators will enforce symmetry;

this can be advertised using the route distribution mechanism.

Should encapsulation be an always-on feature, or only enabled when under attack? The latter would essentially mean that the decapsulator publicly advertises being under attack, potentially inviting other attackers to cause further harm. Our performance numbers, discussed in a later chapter, lead us to believe that the best solution is to always encapsulate. This also serves to publicly advertise which ISPs are being good network citizens and which are not.

3.2.6 Filtering Protocol and Filters

To handle communication between decapsulators and encapsulators we need a new signalling protocol. The primary purpose is to allow a decapsulator to request a filter from an encapsulator, but as we discussed in section 3.2.4, there needs to be more to it than that.

The most important operation is the installation of filters. The protocol should allow the decapsulator to specify what the filter should match, the type of filter, what action to take when a packet matches the filter, and an expiration time.

So what should the format of the actual filters be? From an architectural point of view, a filter in an encapsulator can be anything that stops unwanted traffic with minimal side effects. It is in both sides' interest to install the most specific filters that actually suffice to block the traffic, so long as the encapsulator can maintain sufficient state. In the case of attacks which require a connection to be established, spoofing is not an issue, so specifying both source and destination IP addresses is desirable. In the case of spoofed attacks, the worst case is a flooding attack on a link, where the source addresses can be spoofed and the destination address can be any address beyond that link. In such a case, the decapsulator may be prepared to accept some collateral damage, and request that all traffic from an encapsulator to the decapsulator should be blocked. In between these extremes, we can envisage uses for various combinations of source address prefixes and destination address prefixes.

While the protocol should be flexible enough to accommodate different types of filter, a good starting point would be to initially support three types of filters: a prefix-based IP source address along with a specific destination address; a specific destination address with wild-card source address, whereby the encapsulator will filter all traffic going to that destination; and wild-card source and destination addresses, to address the link flooding attack above. We believe these would cover most of the requirements that might be encountered in the early stages of deployment, and that other policies can be implemented with reasonable performance in terms of these three.

A filtering request should also specify the action to take when a packet matches a filter: besides dropping the packet, the signalling protocol should be expressive enough to at least sup-

port rate-limiting based on a token bucket, even though such capabilities may not be available in all encapsulators.

Finally, filters should be soft-state - they should expire if not refreshed to avoid the filter being orphaned if the decapsulator loses state. Additional ways to expire filters, perhaps based on traffic levels, could be envisaged too, but they are not strictly required to satisfy our basic requirements.

The remaining functions of the signalling protocol, as discussed in section 3.2.4 are:

- Nonce exchange. The signalling protocol cannot use TCP, as this would require too much connection state at a decapsulator under attack. Thus an encapsulator cannot blindly trust the IP address of a decapsulator that requests a filter, as shown in Figure 3.8.5. Instead it validates the decapsulator address by sending a random nonce, and requiring the decapsulator to return that nonce before it will install the filter.

- Request traffic marking. The decapsulator should be able to specify a random number for the encapsulator to include in the encapsulation header, so as to cope with Figure 3.8.4.

- Request encapsulator list. The decapsulator should be able to ask another decapsulator for the list of encapsulators corresponding to a specific client address handled by that decapsulator. This allows for defense against the scenario in Figure 3.8.2.

3.2.7 Evil Bit

Should the "evil bit" apply to all packets, or just to encapsulated ones? This impacts the type of filter needed at the border routers of the server's ISP. If the evil bit is applied only to encapsulated packets, router C1 in Figure 3.5 would have to first separate encapsulated packets from native ones, and then set the evil bit based on the peer the packet came from. Border router E1 would classify all non-encapsulated packets destined for the server to a lower priority traffic class, but it would also classify encapsulated packets as lower priority if they have the evil bit set. The alternative seems simpler: C1 sets the evil bit on *all* packets from ISP A, and E1 installs a single filter based solely on this bit.

Regarding where this bit would be implemented, the second solution requires it to be in the regular IP header, whereas the first is more flexible as it allows the evil bit to be in the encapsulation header, which can be largely of our own design. However, it is unclear whether current backbone routers have sufficient flexibility to set a bit in such a header, or classify based on it. On balance, the simplest solution may be to use a diffserv code point in the regular IP header to signal that the "evil bit" is set.

3.2.8 Routing Protocol Design

We need a routing protocol to distribute the binding between network prefixes and decapsulator addresses to encapsulators worldwide. But this is not a routing protocol in the traditional sense, as it is agnostic to the path that the encapsulated traffic takes to get to the decapsulator. Thus these bindings are much more static than conventional routing information and moreover, the same encapsulation table should be distributed to every encapsulator worldwide. Even through these tables may be very large if our scheme is deployed globally, the actual size of the data is well within the capability of cheap commodity hardware. In terms of a distribution mechanism, the requirement is not dissimilar to that of NNTP[29] or BitTorrent[16], which are both capable of distributing much larger volumes of data relatively reliably to large numbers of hosts worldwide. Basically the requirement is simply for each domain to inject its own prefix-to-decapsulator bindings into the routing system, and for the routing system to then flood those bindings worldwide in a reliable manner.

Obviously it would be simple to hijack traffic for a site if an incorrect mapping could be distributed, so each binding needs to be secured using a digital signature. To do so, some form of public key infrastructure is needed to establish a trust hierarchy. One possibility is to use a hierarchy rooted at ICANN and delegated via the regional registries to ISPs. An alternative would be to use a multiply-rooted hierarchy anchored at the Tier-1 ISPs, delegated via Tier-2s, and so on along pre-existing provider-customer relationships. A third alternative would be to use a model similar to that used for SSL, using arbitrary certification authorities as trust anchors that are unrelated to the routing or address delegation hierarchies. All three are technically viable, but they have different political consequences; which would be best is really outside the scope of this architecture. However, we note that the hierarchy rooted at the Tier-1 ISPs has the advantage that the trust chain matches the existing trust chain of the underlying routing system, making anomalies easier to detect. The disadvantage is that incremental deployment would be harder than the SSL-like model which does not require initial buy-in from the Tier-1 ISPs. In reality a good protocol design might permit any of these options, and the solution used might evolve as incremental deployment progresses.

Once we have signed bindings, we need a distribution mechanism for them. The problem is quite similar to that proposed for pushing DNS data out worldwide[25], an approach called Push DNS. The basic idea is to build a peer-to-peer distribution mechanism, built from infrastructure nodes such as our encapsulators, perhaps supplemented by additional distribution nodes. Peerings between nodes need not follow the normal routing adjacencies; they can simply be configured between any two ISPs that have a business relationship, or who decide to peer just

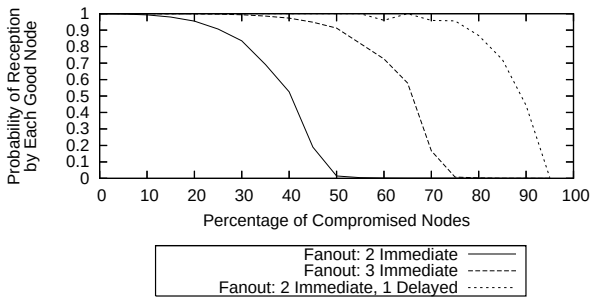


Figure 3.9: *Robustness of routing infrastructure to compromised nodes.*

for this purpose. The only requirement is that the resulting network is connected. These configured peerings would then be supplemented by additional learned peerings which would make the overlay topology richer, so that the overall peer-to-peer network has small-world properties resulting in rapid distribution of data.

As the bindings are signed, every node receiving one should check the signature before passing it on to its peers. Thus there is no possibility for a malicious node to inject bad routing data unless the certificate chain itself has been compromised. Any node receiving bad data from a peer knows that that peer is malicious. Such networks are extremely robust to attack by insiders. The only real attack that is possible is for a malicious node to receive a message but refuses to pass it on. Figure 3.9, taken from [25], shows how robust such overlay networks are. In this simulation, each node passes on a message to two peers; three peers; or first two peers, then (after a short delay) one additional peer who has not yet received the message. This illustrates that the probability of correct delivery is very high right up until the majority of nodes within the network have been compromised.

It is worth pointing out that the nature of the public key infrastructure (both in terms of its internal structure and its position in the network) is independent to how effective this distribution mechanism is. In essence, all that a node in the peer-to-peer network needs is access to the public key infrastructure in order to sign its data and verify data from other nodes.

The major cost in the peer-to-peer distribution network is then to check signatures. Although this could be done in the encapsulators themselves, it is also possible to offload this checking to a fast server at each ISP. Such a server can also cache which signatures it has checked in the past, so only truly new bindings need to be checked, even after a reboot.

Our conclusion is that the distribution of the routes and the checking of signatures on these routes is not only feasible, but also relatively easy and very robust.

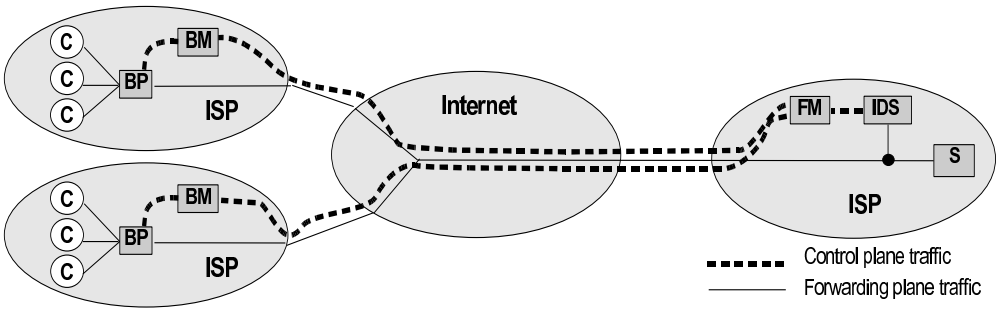


Figure 3.10: *Terminus architecture showing the location of its elements. C stands for client, S for server, BP for border patrol, BM for border manager and FM for filter manager.*

3.3 Terminus Architecture

The architecture we propose in this section, called Terminus, enables a victim to request that certain traffic be stopped close to its sources. While the architecture described in the previous section achieved this goal, Terminus does so while greatly simplifying a lot of the mechanisms.

We start from the assumption that the victim of an attack can tell with reasonable accuracy which traffic is bad. In this section, we refer to the detector as an intrusion detection system (IDS), but in many cases it may be the server itself. We then deploy filtering boxes near sources of traffic, since the size of botnets being reported means it is not possible to defend against large flooding attacks near to the destination, even if the victim is connected to well-provisioned links. For the system as described so far to be viable, the following issues must be addressed:

- Finding the right filtering box from which to request a filter.
- Validating filtering requests to ensure spoofed requests cannot become a channel for attack.
- Preventing spoofed traffic floods from triggering a filter request that blocks legitimate traffic.
- Providing incentives for early adopters, and especially providing incentives to deploy filtering boxes.

From the point of view of deployment, it is critical that the mechanisms will work even when the ISP of the bot and the ISP of the victim are remote from each other and have no prior business relationship. The only form of contractual arrangement that seems viable is that of pairwise service level agreements (SLAs) between neighboring ISPs. Thus any architecture must assume that this is the contractual mechanism from which end-to-end filtering services are built. We primarily use such SLAs to distinguish spoofed traffic to avoid the second issue listed

above. The rest of this section explains the architecture in greater detail, including solutions to all of these issues.

3.3.1 Edge Filtering

Terminus places special control points called *border patrols* (BPs) in ISPs, as close to the sources of traffic as possible (see Figure 3.10). An ISP deploying Terminus (a "*Terminus ISP*") configures its network so that traffic from these sources is forced through border patrols. In this way, each BP can later be asked to install filters for traffic going through it, and, since it is close to the sources, the total aggregate throughput it forwards should be manageable.

With this mechanism in place, the victim of an attack would have to know which BP the traffic came through. In a perfect world, this would be as simple as looking at the source IP address of the malicious traffic and deriving a mapping between this and the correct BP. Unfortunately, because of spoofing, this information cannot be trusted.

This is not to say that currently most sources of attack spoof; they do not. However, this is likely to change if an effective anti-DoS mechanism were deployed. As a result, any such mechanism that does not deal with spoofing in some way runs the risk of quickly becoming obsolete.

Although many ISPs perform ingress filtering to prevent spoofing, enough do not to pose a sizable potential problem; the difficulty lies in that a receiver cannot tell the difference between a spoofed packet and a non-spoofed one, so may incorrectly filter legitimate sources. All is not lost however; the addition of one simple mechanism avoids this problem.

The idea is to use a "true source" bit, similar to the "evil bit" in the previous architecture, in the IP header to mark whether the source IP address field in a packet is in fact the address of the host that originated the packet. As the packet travels from source to destination, Terminus ISPs have their ingress edge routers set or unset this bit depending on whether the packet came from a peering link to a Terminus or legacy ISP, respectively; the routers would know this through pairwise contractual agreements. In this way, if a packet traverses only Terminus ISPs on its way from source to destination, it will arrive with its true source bit set, and its source IP address can be trusted. Of course, Terminus ISPs are assumed to perform ingress filtering, either at their routers or their border patrols.

Figure 3.11 gives a couple of different scenarios illustrating this mechanism. Packets originating at ISP A and going to the server hosted by ISP G will arrive with the true source bit set. Packets from ISP B, on the other hand, will have this bit unset by router E2, since it knows that its link connects to a legacy ISP. Finally, any packet from ISP C or D will arrive with the bit unset, since router G2 knows ISP F to be a legacy ISP.

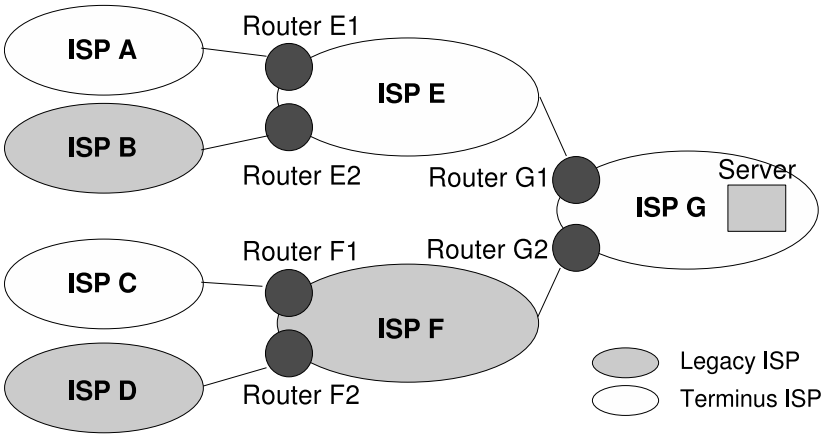


Figure 3.11: *True source bit scenarios.*

Thanks to the border patrols and the true source bit, a victim can now know whether the source IP address in a packet is valid or not. Naturally, the server will only be able to trust the source IP address field for packets that traversed a Terminus ISP-only path, but as deployment progresses this will become the common case.

3.3.2 Filtering Requests

We now have control points (border patrols) deployed on outgoing paths to the rest of the Internet, and a way for a victim to determine where a packet came from (the combination of the true source bit and the IP source address field). When an attack is detected, the next step is to send filtering requests.

Although the IDS or server acting as the detector knows what it wants to filter, it does not know where to send the request. To avoid burdening an already busy system and to avoid revealing the existence of an IDS, we offload this to another system, which we will call a *filter manager* (FM). The IDS is simply configured with the address of its local FM, and sends all its filtering requests there (Figure 3.10).

An FM needs to map an IP address to be filtered to the address of the border patrol handling traffic from that IP address. There are many ways to do this, but our preferred solution is to again use the peer-to-peer flooding protocol from Push DNS to distribute digitally signed bindings to all FMs worldwide.

The size of this "routing" table would certainly be manageable: only one entry would be required per AS, or about 20,000 entries in the current Internet. Each entry could consist of an IP address and a set of prefixes, aggregated as much as possible, representing the clients of the ISP that sit behind border patrols.

Using this mapping table, the FM determines the address of the host at the remote ISP

from which it needs to request a filter. Such a host, called a *border manager* (BM), needs to forward the filtering request to the appropriate border patrol (in a small deployment, the BM could be the same host as the BP; the architecture places no requirements on how this should be implemented). To accomplish this, the ISP could install the necessary mappings of source IP addresses to border patrols, and update them should these change.

Once filters are installed we need a way of removing them. Simplest is to include, along with the filtering request, information about how a filter should expire. The BP then removes the filter when the criteria are met (the expiration could be time-based or even rate-based). When filters are installed and attack traffic subsides, the victim has no obvious way to know if the attack has actually ceased or if it is the filtering mechanism that is being effective. The IDS could of course request the filter be removed and measure the effects, but perhaps a better solution is to provide a way to retrieve filter traffic statistics from BPs. This allows the IDS to explicitly remove unneeded filters, and provides a more flexible tool for the IDS to use as it sees fit. The filtering protocol described in Appendix A supports all of these approaches.

We now have all the basic elements needed to filter an attack. Traffic from clients traverses border patrols and arrives at the server, where a nearby IDS detects the attack, determines the malicious sources, and sends a filtering request to its local filter manager. The FM, in turn, uses the mapping of source IP address to border manager obtained via the peer-to-peer network to send the necessary filtering requests to the appropriate border managers. These, in turn, ensure that the requests go to the appropriate border patrols through which the malicious traffic flows, and where it is finally filtered.

3.3.3 Protecting the Architecture

With such a powerful mechanism in place, care must be taken to make sure that the architecture is not used as a DoS tool in its own right; protecting it from such misuse and dealing with other forms of attacks is the topic of this section.

3.3.3.1 Defending Against Bots at Legacy ISPs

If Terminus were fully deployed, large DDoS attacks could be filtered even if a few legacy ISPs remained. However, during initial deployment legacy ISPs will be the norm rather than the exception. Thus, we need to provide some level of protection against attacks by sources hosted by legacy ISPs.

To this end, we can make use of the true source bit already described. This not only denotes that the IP source address is valid, but it also says that the packet originated at and has traversed Terminus ISPs. It makes sense for a Terminus ISP to reward other Terminus ISPs, and so we can

install diffserv classifiers at the edge routers of the destination ISP (or indeed other Terminus ISPs on the path if they wish to do so), sending packets that have the true source bit set to a higher priority queue. As a result, packets from legacy ISPs will always get lower priority and, during an attack, potentially little or no service. It now becomes clear that although we always refer to the true source *bit*, in reality we implement it as a new *diffserv codepoint*, so can use the existing diffserv machinery in all modern routers.

3.3.3.2 Validating Filtering Requests

As described so far, an attacker could contact a border manager and request a malicious filter. A nonce exchange suffices to avoid this. On receipt of a filter request, the border manager sends a random nonce back to the filter manager, and only installs the filter when it gets the nonce echoed back. This serves to validate that the IP address of the FM is not spoofed (of course this is not strictly necessary if the true source bit is set in the filtering request, but as Internet paths are asymmetric we cannot count on this being the case, and the extra validation is cheap).

Validating the FM's IP address is not sufficient though: it is also necessary to validate that this particular FM is authorized to request a filter for this particular destination IP address. In essence we need the reverse mapping table from the one used by the FM to discover the BM's address. The same peer-to-peer distribution of digitally signed mappings can be used to distribute these reverse mappings too. Likely the certification authorities for these signatures will be the Regional Internet Registries (RIRs), as they already handle IP address allocations to ISPs. It is worth noting that the nonce exchange will not stop an attacker on the path between the BM and the FM; however the additional risks are minimal as a compromised router can already filter the traffic by simply dropping it.

With this in place, upon receiving a filtering request the border manager will inspect its source IP address. If the mapping between this address and the destination address of the actual filter exists in the set of mappings distributed using the peer-to-peer network, then the BM will issue a nonce. This nonce will reach the FM, which will echo it if it had, in fact, issued a filtering request. Finally, the BM will contact the appropriate border patrols to block the unwanted traffic.

3.3.3.3 Triggering Requests Through Spoofing

Although the architecture ensures that filter requests come from legitimate parties, it might still be possible for an attacker to spoof client traffic to trigger a filter against an unsuspecting legitimate client. The list of possible attack scenarios have to do with the location of the attacker with regards to the victim. Only five such scenarios exist:

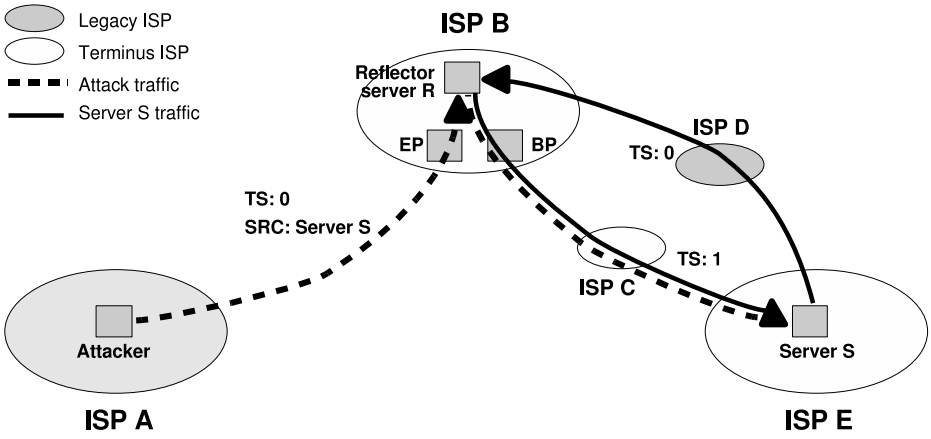


Figure 3.12: *Reflector attack scenario. TS stands for true-source bit, SRC for IP source address, EP for egress patrol and BP for border patrol.*

(i) The attacker is in a legacy ISP that allows spoofing.

(ii) The attacker is in a legacy ISP that performs ingress filtering.

(iii) The attacker is in a different Terminus ISP from the real client.

(iv) The attacker is in the same Terminus ISP as the real client but behind a different BP.

(v) The attacker is behind the same BP as the real client.

In the first scenario, the attacker can spoof the client's address. However, the attacker's packets will arrive with their true source bit unset. The filter manager must err on the side of "do no harm" and only issues a filtering request when the true source bit is set, relying on low diffserv prioritization when the bit is not set. The next two scenarios are impossible: an attacker from an ISP that performs ingress filtering simply cannot spoof the address of a client in a different ISP. The attack described in the fourth scenario is easily preventable by either performing ingress filtering at the BPs or by ensuring that the ISP uses the true source bit internally. In the last scenario, the BP cannot tell traffic from the attacker and the victim apart. In essence, the problem is a local one, and the ISP can use ingress filtering or other local sanctions.

3.3.3.4 Reflection Attacks

In a reflection attack, the attacker spoofs a high rate of requests using the victim's IP address, and sends them to innocent third-party servers; the response flood then overwhelms the victim. A typical reflector might be a DNS server, and the motivation is to amplify the attack as responses from the server are larger than the requests sent by the attacker.

For the most part, reflection attacks do not trouble Terminus. If the attacker is at a Terminus ISP, spoofing is not possible. If the reflector is not at a Terminus ISP, or the path from the reflector to the victim is not fully Terminus-enabled, then the attack traffic does not have the true source bit set, getting low priority at the ISP of the intended victim. However, there is a single deployment combination that is problematic, as shown in Figure 3.12. The issue arises when the attacker is at a legacy ISP, so can spoof, and the reflector is at a Terminus ISP, so traffic from the reflector to the victim arrives with the true-source bit set. In this case the reflector has "promoted" low-priority attack traffic to a higher priority. This scenario needs special treatment.

The solution requires an additional system called an *Egress Patrol* or EP. In practice, EPs and BPs will almost certainly be the same systems; the difference is primarily that EPs filter traffic inbound to customer hosts, whereas BPs filter outbound traffic. All traffic to servers that might be used as reflectors is directed through an EP.

With regards to Figure 3.12, the process begins when an IDS near server S detects a reflection attack (responses are coming for requests that were never sent), and alerts S's FM. The FM cannot simply request that the attack is filtered, because if it did, then the responses to any requests from S to R would also be filtered. Thus an astute attacker might be able to cause Terminus to block essential communications.

As inter-domain paths are frequently asymmetric (at least for commercial networks, refer to [26]), S's FM does not know whether or not the path to ISP B is Terminus-enabled. The appropriate response depends on knowing this, so the FM sends a conditional filter request of the form:

if filter request packet arrives with $TS = 1$ **then**

At EP, block traffic from S to R where $TS = 0$

else

At BP, set $TS = 0$ on traffic from R to S

end if

The reasoning behind this is as follows:

- If the filtering request packet arrives with TS = 1, ISP B can use the TS bit to distinguish packets that are *actually* coming from S from those coming from A. In this case, we ask the EP to block traffic whose source IP address is S whenever TS = 0, dropping all of the attack traffic.

- If, on the other hand, the filtering request packet arrives with TS = 0, ISP B has no way to tell which packets originated at S and which ones at A. Without adding a lot of additional

mechanism, the best that can be achieved is to avoid promoting the attack traffic, hence the request for a filter to clear the TS bit.

It is possible to handle the latter corner case more effectively by resorting to tunneling, but on balance it seems better not to add too much additional mechanism, but rather simply to encourage wider Terminus deployment.

3.3.3.5 Protecting Terminus' Components

For Terminus to be successful, all of its components must be robust against attack. Attacking border patrols, for instance, could deny traffic from clients from reaching a server. Under full deployment, there will be a significant number of BPs, and so DoSing a server by stopping client traffic would prove very difficult at best. During initial deployment, however, there might only be a few BPs deployed and the attack might be effective. The solution is simple: do not advertise the BPs' address prefixes externally via BGP. If they are not externally reachable they are not susceptible to attack; EPs can be protected in the same manner.

Border managers, on the other hand, do have to be externally visible in order to receive filtering requests. However, these boxes are not on the fast path, and so can devote all their resources to the control protocol. The BM implementation described in Chapter 5 can not only service requests at a fast rate, but also ensures that no state is held for a client before it has responded to a nonce. In the end, overloading a BM with requests only prevents filter installation. To allow a bot behind a BP managed by such a BM to continue an attack requires many bots to DoS the BM; this simply is not a good return on investment for the attacker.

One final element of the architecture that might be targeted is the filter manager. Again, this box is not on a fast path, and so it can devote all its resources to filter requests. More importantly, its traffic is constrained: there are a limited number of IDS systems from which it should accept requests, and the path for nonce requests from BM to FM will always be Terminus-enabled (or we would not have requested the filter in the first place).

3.4 Diff Serv

To accomplish their goals, the last two architectures rely on a bit in the packets' headers along with some mechanism to prioritize packets based on this bit's value; the aim is to be able to cope with spoofing and initial deployment scenarios where not all ISPs have deployed the architectures. The Edge-to-Edge architecture, for example, calls such a bit the "evil" bit and uses it to mitigate attacks originating in legacy ISPs: such traffic cannot be blocked near its sources since there are no encapsulators deployed there; instead, the evil bit is used to give lower priority (at the destination's ISP) to all traffic originating at or traversing legacy ISPs.

Since only legacy ISPs can spoof traffic, this measure also serves to mitigate spoofing.

The Terminus architecture uses a similar, so-called “true source” bit to denote whether a packet’s IP source address could have been spoofed or not; packets whose true source bit is unset are given lower priority at the destination’s edge routers, and no filtering requests are issued for traffic arriving with this bit unset in order to prevent using the architecture as an attack tool by filtering an unwitting victim’s traffic. Further, the true source bit is used to prevent reflection attacks where the reflector is at a legacy ISP or the path from the attacker to the reflector is a legacy one.

As mentioned, one clear candidate to implement the evil and true source bits would be diffserv, since this mechanism already exists in current routers. However, it is worth pointing out that the architectures do not need diffserv to be deployed: all they require is the use of a single bit in the IP header and the ability to prioritize traffic at edge routers based on that bit. In addition, the deployment incentive is strong, since the deploying ISP is the one hosting the victim under attack. Finally, note that since these bits are there to mitigate problems during initial deployment, the importance of these mechanisms will diminish as deployment progresses.

Chapter 4

Baseline Evaluation

The performance of a solution against Denial-of-Service attacks is crucial. A defense mechanism that cannot cope with significant load can be rendered useless by an attacker, regardless of how clever this mechanism may be. As a result, solutions should be validated not in simulation but by using real hardware.

While the last two architectures we presented in the previous chapter have the advantage of filtering at the edges, we must still test their performance to ensure that they can deal with large loads. As with any performance evaluation, repeatability is key, and so using a controlled network testbed is a logical step. However, creating realistic attack scenarios in a testbed is problematic, since whatever scenarios are chosen, they could never be general enough to reflect real-world diversity. Despite this, it is possible to test each of the components of the architecture individually to see how they behave under heavy load and pathological cases, and thus provide some level of confidence with regards to the architectures' overall performance.

Ideally we would like the mechanisms described so far to be implemented in hardware, perhaps as part of a router platform. In reality, however, and especially during the early deployment stages, it is unlikely that commercial vendors will adopt the approach without having seen some level of real-world deployment. Consequently, we have opted to implement the solutions using off-the-shelf hardware to show their feasibility, but also to provide a cheap protection platform in hopes of motivating further deployment. In this chapter we present an in-depth baseline of the hardware including the testbed used, while in the next chapter we focus on the performance of the elements of the architectures.

4.1 Testbed

We conducted all performance experiments on the *Heterogeneous Experimental Network* (HEN)[57]. HEN is a testbed designed for carrying out network experiments in a controlled, highly-configurable and repeatable environment. It consists of over 100 computers of various

Model	Processor	# Procs	Memory (in GB)	Interfaces	Bus Type
Dell 1950	Intel Xeon 5150 2.66GHz Dual Core	2	2	2 x Intel 82571EB Dual Port	PCI-e
Dell 2950	Intel Xeon X5355 2.66GHz Quad Core	2	8	3 x Intel 82571EB Quad Port	PCI-e

Figure 4.1: *Capabilities of testbed computers used.*

capabilities, each containing at least four network interfaces. In addition, each of these is booted over the network, allowing users to change the operating system of any or all computers in an experiment in a matter of minutes; this also facilitates kernel work, since fixing a crash is as simple as rebooting the machine and telling it to boot a previous, working kernel.

At the core of the testbed lies the main experimental switch, which consists of a Force10 E1200[20] with over 500 gigabit Ethernet ports. The switch can be programatically configured to create VLANs, allowing researchers to setup even complex network topologies in moments. In addition, the E1200 is non-blocking, so there is no danger of an experiment's outcome being affected by another experiment.

Table 4.1 shows the capabilities of the various computers we used in our experiments. In general, we used the more powerful machine (the Dell 2950) to perform the actual forwarding and filtering, while using the Dell 1950s to generate and count traffic. As can be seen from the table, the experimental interfaces in all the computers were 1Gb Intel cards either with 2 or 4 ports, connected to PCI-express buses.

We used Linux 2.6 for all our testing, using a modified version of the e1000 driver that contained polling extensions. To generate, forward and count packets we used the Click modular router platform[40]. Click has several advantages over native Linux forwarding and other packages. First, it provides excellent performance when run in kernel mode with the polling driver. Second, it is easily configurable via its own description language. Finally, should the needed functionality not be included in the default distribution, Click can be extended by adding user-created modules.

All performance measurements reported here are the result of averaging the results of three different runs. The reason behind this is that initial tests revealed that using more runs (as many as 10) had negligible impact in the values of the resulting figures. Further, the values for each of the runs differed very little, thus justifying the use of the averages reported in this thesis.

4.2 Theoretical Maximum Rates

At a high-level, the filtering elements of the architectures are not quite routers but rather "forwarders", devices that receive traffic on one interface and send it out on another interface,

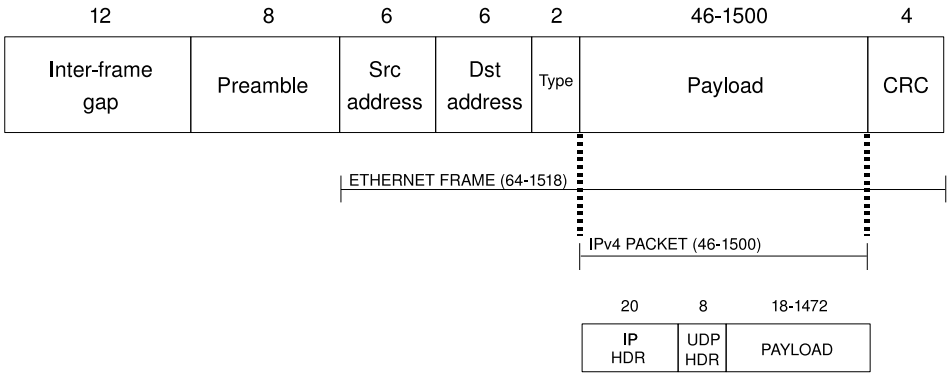


Figure 4.2: *Ethernet frame with UDP payload. All field sizes are in bytes.*

regardless of what the traffic's destinations are. Like any other device that processes packets, their performance is highly affected by the size of the packets that they receive, with minimum-sized packets causing the greatest performance hit. As a result, even though we will present results for a range of packet sizes, our focus will be on small ones.

Before presenting actual base line figures, it is worth looking at what the theoretically-maximum rates for Gigabit Ethernet are for various packet sizes. Besides the overheads incurred from the frame's header (12 bytes for the source and destination MAC addresses and 2 bytes for the type field), an Ethernet frame contains an 8-byte *preamble* to allow time for the receiver to synchronize the receive data clock to the transmit data clock; a 12-byte *inter-frame gap* used to allow the signal to propagate through to the receiver electronics at the destination; and a 4-byte CRC (see Figure 4.2). This means that, for instance, a minimum-size packet will actually have 84 bytes when it is on the wire, and a maximum-size one 1538 bytes.

As a result of these overheads, the actual rates achievable are less than 1Gb/s, and decrease with the size of the packets. Table 4.3 shows the theoretical maximum rate for Gigabit Ethernet for various packet sizes. It is also worth noting that for our experiments we used UDP traffic since not only it is cheaper for an attacker to generate, but it is also easily spoofable and so more attractive from his point of view.

4.3 Generators and Counters

To test the performance of the various elements of the architectures we need both packet generators and counters. In addition, these must be able to process packets at line-rate for all packet sizes, otherwise the forwarding figures might be skewed by bottlenecks in these devices. For the experiments in this section we used Dell 1950 computers as generators and counters, since we have a reasonable number of them in the testbed. Each of these had four network interfaces, and so the goal was to see how many of these interfaces could be simultaneously used to handle

Payload size (bytes)	Frame size (bytes)	Size on wire (bytes)	Theo. Max (in Kpkts/s)	Theo. Max (in Mb/s)
46	64	84	1488	762
100	118	138	906	855
128	146	166	753	880
200	218	238	525	916
500	518	538	232	963
1000	1018	1038	120	981
1500	1518	1538	81	987

Figure 4.3: *Maximum theoretical rates for Gigabit Ethernet for various packet sizes. Kpkt/s stands for thousands of packets per second.*

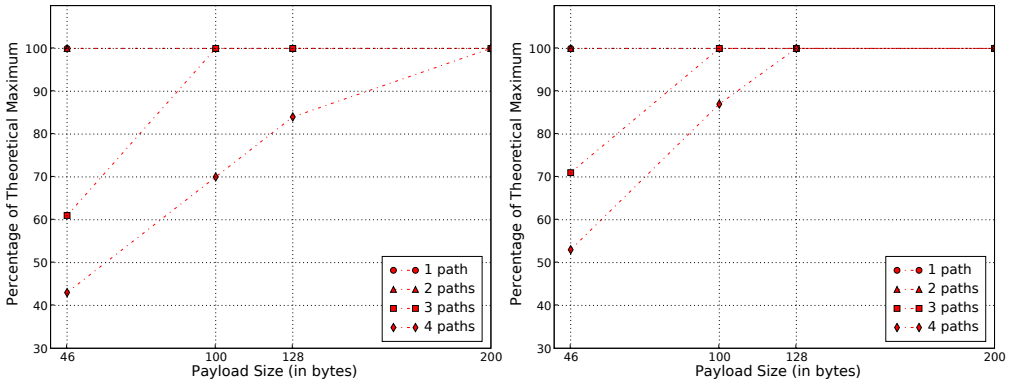
(a) (b)

Figure 4.4: *Network topologies, G stands for generator, C stands for counter and black squares denote network interfaces. (a) Generator performance test. (b) Counter performance test.*

packets at line-rate for all packet sizes; we again relied on Click to both generate and count packets.

To test generator performance we set up the topology shown in Figure 4.4(a). The reason for using two interfaces for each of the counter computers is that, as we will show later, this is the maximum number of interfaces that they can receive packets on without introducing bottlenecks. We ran tests for a set of different packet sizes, ranging from minimum to maximum size. The sizes we picked are skewed towards the smaller end, since it is these we are most interested in in terms of performance. In addition, we ran three tests for each of these sizes, measuring rates in packets per second (pps) at the two counting computers, and averaging the results. Finally, we repeated these tests for one, two, three and four paths, where we define a path to be a connection from a generating interface to a counting one.

As can be seen from the results shown in Figure 4.5(a), the generator can send packets at line-rate for all sizes for one and two paths (note that only small packet sizes are shown in the graph for simplicity's sake; the generator can send at line-rate for all other sizes and all four paths). However, attempting to send over more than two paths results in bottlenecks at least for minimum-sized packets, the size we are interested in since it puts the greatest strain on routers. Consequently, we only use two interfaces to generate traffic for all subsequent tests.



(a) (b)

Figure 4.5: *Performance as a function of theoretical maximum rate for small packet sizes. (a) Generator performance test. (b) Counter performance test.*

Figure 4.4(b) shows the network we set up to test counter performance. The previous experiment showed that a generator can send packets out of at most two interfaces without bottlenecks, and so we had to use two generators to ensure that any bottlenecks would come as a result of the counter computer. Similar to the previous results, Figure 4.5(b) shows that the counter can only receive at line-rate for all packet sizes for two interfaces, but no more. Again, from this point on we limit receipt of packets on these computers to two interfaces.

It is worth pointing out that we performed both the generating and counter tests mentioned so far on single-processor kernels. However, we also ran tests using a symmetric multiprocessing (SMP) kernel and multi-threaded Click and also discovered bottlenecks for minimum-sized packets, albeit smaller ones.

4.4 Linux Forwarding

As mentioned earlier, Click provides several advantages over Linux when it comes to packet processing. However, performance is crucial, and so we need to ensure that Click actually outperforms Linux. For the tests in this section we used the same Dell 1950s to generate and count traffic, and a Dell 2950 running an SMP Linux kernel to perform the actual forwarding (this is the same computer that we will use for most of the forwarding tests in this thesis). For these tests we used the Linux NAPI driver, equivalent in functionality to polling driver used in the Click tests. The network topology is shown in Figure 4.6.

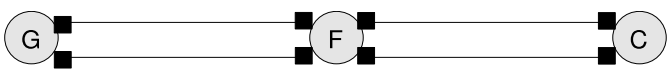


Figure 4.6: *Network topology for Linux forwarding performance test. F stands for forwarder.*

The results in Figure 4.7 show that Linux struggles to forward small packets at line-rate even when only one path is used. Repeating the test with two paths yields worse results (even a payload size of 200 bytes cannot be handled at line-rate), showing that Linux's performance does not scale well with the number of interfaces. In the next section we will show that Click does not suffer so severely from these issues, largely outperforming Linux regardless of how many paths are used.

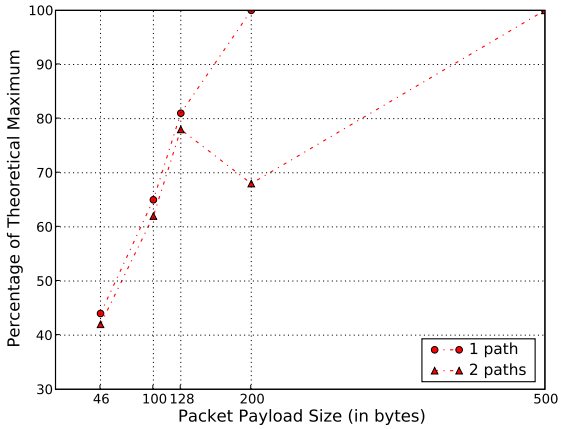


Figure 4.7: *Linux forwarding performance as a function of theoretical maximum rate for small packet sizes.*

4.5 Click Forwarding

Click is a flexible and extensible architecture for building performant network devices. A Click configuration is made up of packet processing modules called *elements*. These elements perform simple network functions such as queueing, routing, classifying and interacting with network interfaces. A configuration consists of a directed graph with the elements as vertices and packets flowing along its edges, and Click provides its own language for declaring such configurations.

Click schedules the CPU using a task queue. When an element is scheduled, it performs its packet-processing task and calls a downstream element. This process repeats until the packet in question is explicitly stored, dropped or sent either to the host or over the network. Thus, the location of queue elements in the configuration determines how CPU scheduling is performed: queues that are several elements away from input device elements result in the computer having to do a significant amount of work before it can process the next packet. Because of this model, the Click tasks (or scheduling unit) in a router consist of the set of elements from the input network device to the queue, and those from the queue to the output network device.

In Linux, Click can operate both in user level and in the kernel. In addition, Click works

with the regular Linux network drivers, but provides a customized version of the Intel e1000 driver with polling extensions. Since we are concerned with performance, we use Click in kernel mode with the polling driver for all our experiments.

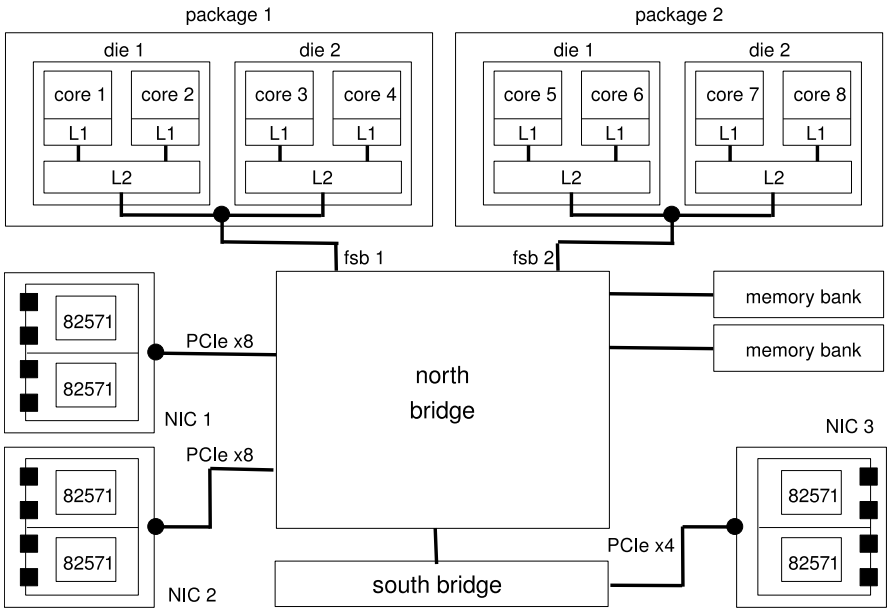


Figure 4.8: *Dell 2950 architecture.*

While we use Dell 1950s to both generate and count packets, we rely on a more powerful Dell 2950 to actually process and forward them (Figure 4.8). The computer has two Intel X5355 processors, each consisting of two dies and each of these, in turn, containing two CPU cores at 2.66GHz and 4MB of level 2 cache shared among the two cores. Further, each core has 32KB of level 1 instruction cache and another 32KB of level 1 data cache. The processors are connected to the north bridge (Intel 5000X chipset) via dual front side buses running at 1.3Ghz. In addition, the computer has 8GB of 667MHz memory. In terms of networking, the system has two PCIe x8 slots and one PCIe x4 slot, each hosting an Intel Gigabit quad-port card with a PCIe x4 interface. Each of these cards contains two Intel 82571 chipsets, each handling two of the ports on the card.

To run experiments we need to come up with Click configurations to perform the actual packet forwarding. For this section, we used two such configurations. The first consists of a level-2 device that receives packets on one interface and emits them on another one, used to provide a very basic performance baseline (Figure 4.9(a)). In the second configuration the device receives packets on an interface, classifies them as IP or non-IP, strips the former's MAC header, and then forwards all of them onto a second interface (Figure 4.9(b)). In effect, this is a minimal level-3 (IP level) device that, for the moment, performs no action on IP packets. We

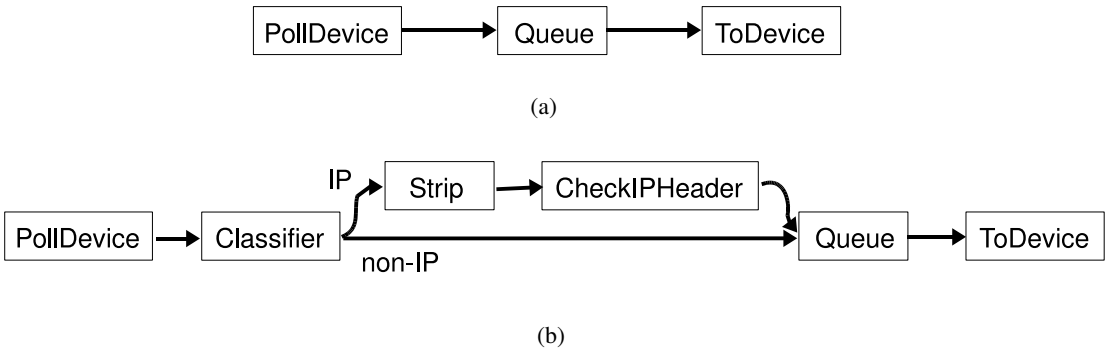


Figure 4.9: *Click configurations for baseline performance tests for a single forwarding path. (a) Level-2 forwarder test. (b) Level-3 minimal forwarder test.*

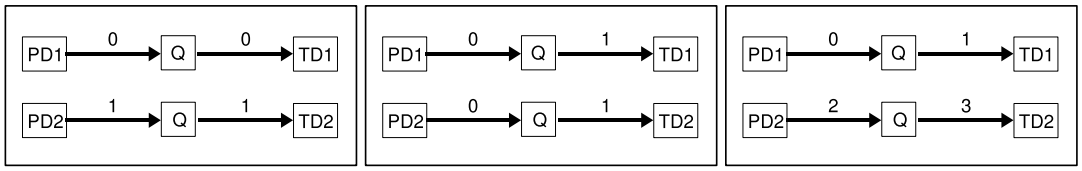
will add our own filtering elements to this basic configuration in later sections of this chapter, so it provides a good measure of baseline performance.

4.5.1 Multi-threaded Click and Affinities

Running the configuration shown in Figure 4.9(b) with two forwarding paths using the topology shown in Figure 4.6 results in poor performance, since Click is by default single-threaded and cannot, as a result, take advantage of the multiple processor cores in the system. Fortunately, Click provides multi-threading in the form of two elements, `BalancedThreadSched` and `StaticThreadSched`. `BalancedThreadSched` assigns threads to portions of a configuration by attempting to minimize variance in load. `StaticThreadSched`, on the other hand, allows the user to statically assign threads to specific sections of forwarding paths. More specifically, in our configurations the element assigns either a `PollDevice` or a `ToDevice` to a thread ID number. In the case of a `PollDevice`, this means that the thread will execute packet processing from this element all the way to the queue element; for a `ToDevice` it means that the thread will process packets from the queue all the way to the output element.

For the `StaticThreadSched` element, a question arises as to how many threads to declare as well as to how to assign the poll and todevices to them. To investigate these questions we designed three different assignment scenarios. In the first, one thread is used for a full forwarding path, for a total of two. In the second, all polldevices are assigned to one thread, while all todevices are assigned to a second thread. Finally, in the third scenario each of the parts of the forwarding paths (from polldevices to queues and from queues to todevices) are assigned to separate threads, for a total of four threads (Figure 4.10)

A further question is how to assign these threads to the various processor cores in the system. While the simplest choice is clearly to let Linux handle this, we can also set the affinity of a Click thread. In the case of the Dell 2950, this opens up a number of possibilities, since threads can be set to run on different cores but on the same die, on different dies in the same



(a) (b) (c)

Figure 4.10: *Scenarios for assigning PollDevices and ToDevices to Click threads for two forwarding paths. Numbers represent thread IDs. (a) Forwarding path handled by one thread. (b) Input and output part of forwarding path handled by separate threads. (c) All parts of the forwarding paths handled by separate threads.*

package, or on different packages.

To get a sense for which combination of Click elements to threads and threads to processor cores yielded the best performance numbers we ran four experiments, covering the cases mentioned in the previous paragraph: letting Linux handle affinity, assigning threads to cores on the same die, assigning them to cores on separate dies but the same package, and assigning them to cores on separate packages. For each of these four cases we ran the three scenarios in Figure 4.10 (where applicable). For all of these we assigned at most one thread per processor core, and we set the affinity of the *init* process (pid 1) to a "free" core, so that no processes would run on cores handling Click. Further, we used the Click configuration shown in Figure 4.9(b).

Figure 4.11(a) shows performance results when letting Linux handle affinity. As expected, Scenario 1 yields the best performance (89% of the theoretical maximum for 64-byte packets), since all packets on a forwarding path are handled by the same core, maximizing level 1 cache hits. Conversely, Scenario 2 performs rather poorly (32%): not only does Linux tend to assign the two Click threads to different cores (causing the cache to thrash), but the device is further bottlenecked by the fact that only one core handles all input for both forwarding paths and another one all output. Scenario 3 still suffers from the first performance hit but not from the second one, since it uses four processor cores. Consequently, its performance is between that of the two previous scenarios, 71% for minimum-sized packets.

In the second set of experiments we assigned threads to separate cores on the same die (Figure 4.11(b)). Scenario 1 yields similar results as in the previous experiment, since each forwarding path is still being executed by a single processor core. Scenario 2 again results in poor performance (58% of the theoretical maximum), but shows improvement from the previous experiment. One plausible explanation for this is that while the level 1 cache is being thrashed, the level 2 cache, which is shared by both cores, is experiencing a good hit rate; in the previous experiment, Linux would schedule threads on cores regardless of whether the latter were on the

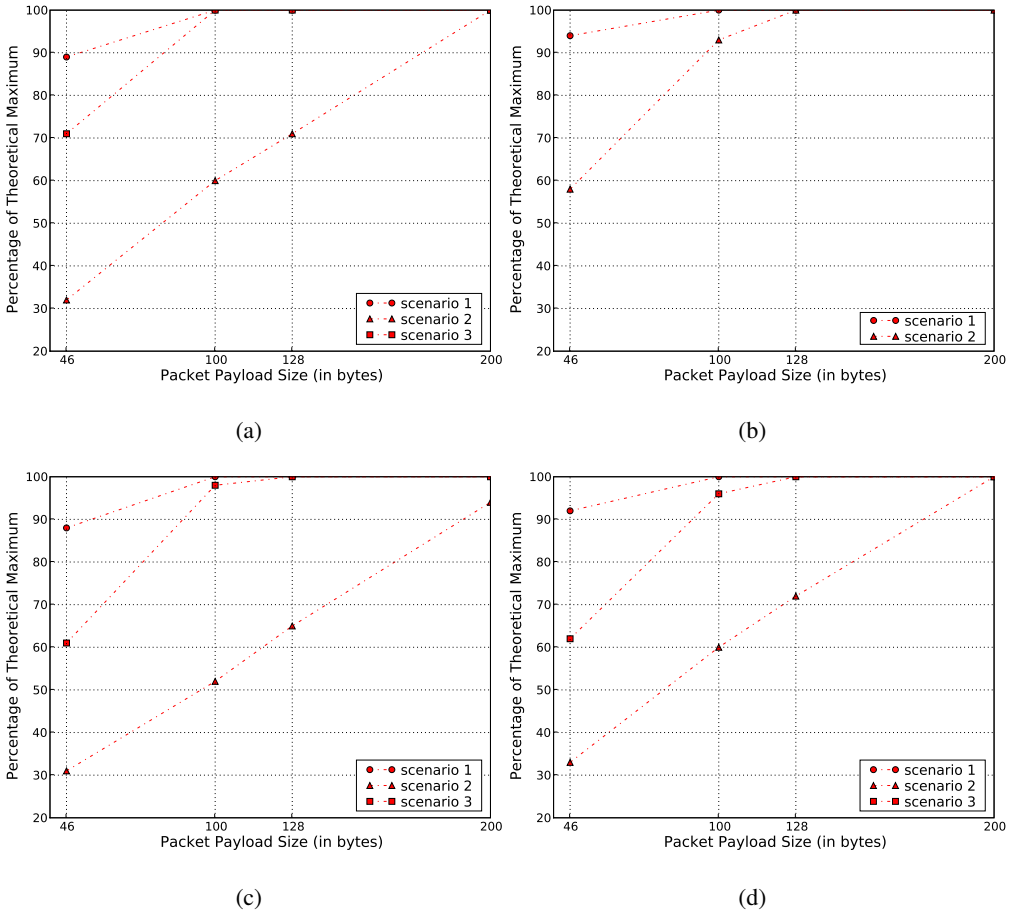


Figure 4.11: *Performance as a function of theoretical maximum rate for small packet sizes and different affinity policies. (a) Affinity handled by Linux. (b) Static affinity, threads on same die. (c) Static affinity, threads on different dies but same package. (d) Static affinity, threads on different packages.*

same die, accounting for the difference. Finally, Scenario 3 does not apply to this experiment since it involves four threads that could not fit individually into two cores.

In the next experiment we assigned threads to cores on the same package but on different dies (Figure 4.11(c)). Unsurprsingly, Scenario 1 yields the same results as before, since forwarding paths are still being ran entirely on a single processor core. Scenario 2 yields 31% of the theoretical maximum for minimum-sized packets. This is also expected, since it is essentially the same experiment as when Linux assigned threads to processors on separate dies (resulting in the 32% figure given above). Scenario 3 yields a value of 61%, slightly worse than in the previous experiment, since presumably now even the level 2 cache is being thrashed.

For the final experiment we assigned threads to cores on separate packages. Once more, Scenario 1 yields the usual and expected performance. Scenarios 2 and 3 also result in similar values as the previuos experiment, 33% and 62% of the theoretical maximum for minimum-sized packets, respectively. This is normal, since in both this experiment and the previous one

scenario	L1 cache misses per cycle	L2 cache misses per cycle
packets do not change cores	0.0070	0.0016
packets change cores on same die	0.0129	0.0013
packets change cores on different dies	0.0119	0.0077

Figure 4.12: *L1 and L2 cache miss counts for various forwarding scenarios. The figures are the average of the values across processor cores.*

the cores being used did not share a level 2 cache, having to access main memory to exchange packets.

To confirm these theories about cache misses we decided to take a closer look at the CPUs. The Intel Xeon processors in the test system provide per-core registers called performance monitoring counters that can be used to measure a wide range of statistics, including cache misses, number of retired instructions and clock cycles. In order to instrument these registers we implemented *evtmonitor*, a Linux kernel module that uses the /proc virtual file system to provide an interface to them. For each run of an experiment we reseted the counters, started them when the first packets were generated, and stopped them when all packets had been either forwarded or dropped. Since the CPU cores are otherwise idle and because we are dealing with millions of packets per second, we assume that the counters provide a good measure of the performance of the forwarding process. Finally, we set the affinity of each Click thread so that it ran in its own core, thus isolating the performance of each core.

With this in place we measured cache misses under three scenarios: the optimal case where packets do not change cpu cores, another one where packets change cores on the same die and finally one where packets change cores on different packages. For each of these we forwarded minimum-sized packets on two paths while measuring the number of L1 data and L2 cache misses, as well as the number of cycles where the cores were active.

Figure 4.12 shows the L1 and L2 cache misses per cycle for each of the three scenarios, confirming our theories. As expected, the first scenario, which yields the highest forwarding rates, has the lowest counts for both L1 and L2 cache misses. In the second scenario, where packets change cores on the same die, the L1 count naturally rises while the L2 count is similar to that of the first scenario, a result of the two cores on the die sharing the same L2 cache. Finally, the third scenario has a similar L1 count as in the previous scenario since packets are still changing cores. However, the L2 count rises, since the two cores are on separate packages, forcing accesses to main memory.

The clear conclusion from all of these experiments is that to obtain the best forwarding

performance one core should handle an entire forwarding path. In terms of setting the affinity, Linux does a fairly good job of this (in the one path-per-core scenario), and so we do not explicitly set the affinity of Click threads in subsequent experiments.

4.5.2 Forwarding Performance

We now have a clear understanding of how to assign Click elements to threads and how to set their affinities in order to obtain the best performance results. Going back to the anti-DoS architectures, we need to build fast devices that receive packets on one interface, apply filters to them, and emit them on another interface. Each of these forwarding paths requires two interfaces, and the hope is that their performance will scale with the number of interfaces and processor cores in the system. In the case of the Dell 2950 there are 12 interfaces available for experiments (the motherboard interfaces are used for management purposes), for a total of 6 forwarding paths.

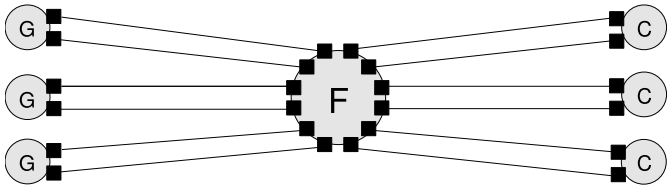
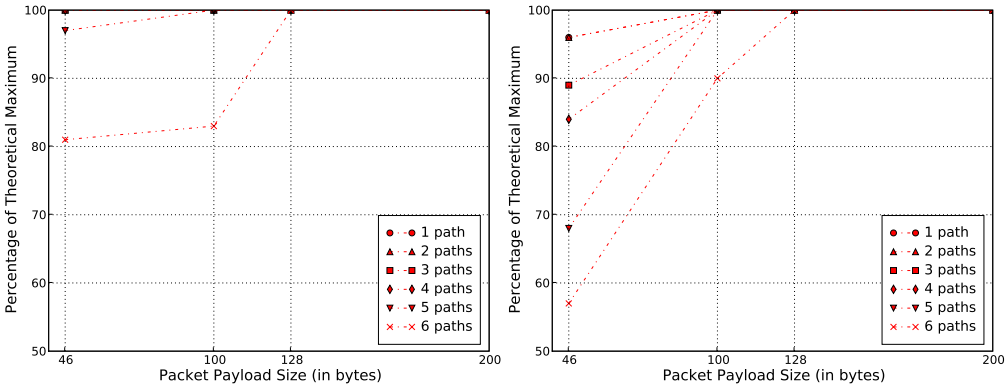


Figure 4.13: *Network topology for Click forwarding performance tests.*

In the next experiments we investigate the performance of the computer for various numbers of forwarding paths. To do so, we use the network topology shown in Figure 4.13, assigning each forwarding path to a thread, and letting Linux handle affinity. For the first experiment we use the level 2 forwarder Click configuration shown in Figure 4.9(a), followed by a performance test of the level 3 minimal forwarder in Figure 4.9(b).

What is immediately apparent from the results shown in Figure 4.14 is that the device can handle traffic at line-rate for all forwarding paths and for all but the smallest packet sizes. In the case of the level 3 forwarder, this results, for example, in a rate of approximately 4.6 Gb/s (about 5 million packets per second) for six paths and a payload of 100 bytes; the figure climbs up to about 6 Gbps for maximum-sized packets, the theoretical maximum.

What is also evident from the graphs is that with 64-byte packets the system becomes bottlenecked as forwarding paths are added. Placing the forwarding rates for minimum-sized packets and different number of forwarding paths on a table illustrates this even more clearly (Figure 4.15). Adding the fifth and sixth forwarding paths does not significantly increase forwarding performance for the device; in the next section we look into the cause of this bottleneck.



(a) (b)

Figure 4.14: *Performance as a function of theoretical maximum rate for small packet sizes and different Click configurations. (a) Level 2 forwarder performance test. (b) Level 3 minimal forwarder performance test.*

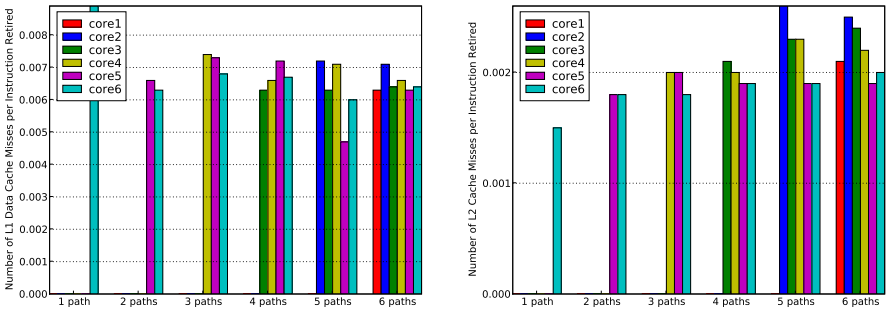
# Paths	Rate (in Mpps)
1	1,432
2	2,843
3	3,961
4	4,975
5	5,068
6	5,094

Figure 4.15: *Forwarding rates (in millions of packets per second) for minimum-sized packets.*

4.5.3 Performance Bottleneck

Packet forwarding is a complex operation involving several hardware components: the CPUs, the front side buses, the memory, the memory controller (also known as the north bridge), the I/O bridge (the southbridge) and the network cards (see Figure 4.8). Clearly any or even a combination of these could be the cause of the bottleneck, and so we will investigate each of these to try to identify the culprit. It is worth noting that we conducted all experiments in this section using the 12-interface setup shown in Figure 4.13 and that all packets were minimum-sized ones.

As a first step, we decided to see whether the CPU cores where the source of the bottleneck. Since one core is able to forward one path at line-rate, we would expect six cores to be able to forward six of these paths, specially since we made sure that packets do not cross forwarding paths, thus reducing cache thrashing issues. Even so, to verify that the caches were not the source of the problem we again used the *evtmonitor* kernel module and the CPUs' performance



(a) (b)

Figure 4.16: *Ratios derived from Intel Xeon's performance monitoring counters for different number of forwarding paths. (a) Number of L1 data cache misses per instruction retired ratio. (b) Numer of L2 cache misses per instruction retired ratio.*

counters. As shown in Figures 4.16(a) and 4.16(b), we measured the number of level 1 and level 2 cache misses for each core and found that these numbers remain almost the same (the graphs show very small differences of 1 part in 1,000, essentially no change) regardless of whether the system was bottlenecked or not; this shows that poor cache performance is not the bottleneck.

Having eliminated the CPU cores as the cause, the next possible source of conflict are the front side buses, each of which is connected to one of the two CPUs in the system. Once again, we made use of performance monitoring counters to derive ratios that directly measure activity on these buses. The two most important ratios measured the percentage of bus cycles used for transferring data, and the percentage of bus cycles during which new transactions could not start because the bus was busy. As expected if we assume that these buses are not the bottleneck, these percentages were quite small for both buses and saw little change as we added forwarding paths (we are not listing the actual numbers here for brevity's sake). We repeated the same experiment when using six forwarding paths but assigned these to various combinations of CPU cores and obtained similar results, thus concluding that the front side buses were not bottlenecked.

Having eliminated the CPUs and the front side buses, we are now left with the PCIe bus[1], the memory controller or the memory as possible culprits. In order to investigate the first, we extended the e1000 driver to take advantage of PCIe statistics registers that are available in each of the Intel network cards. These can measure a wide range of events, including the number of transmitted and received packets, completion latencies and the number of lower-layer flow control packets.

Before delving into these counters, we want to determine where packets are being dropped

[1]The term bus is a misnomer, since PCIe consists of a network of serial lanes connected by a crossbar switch.

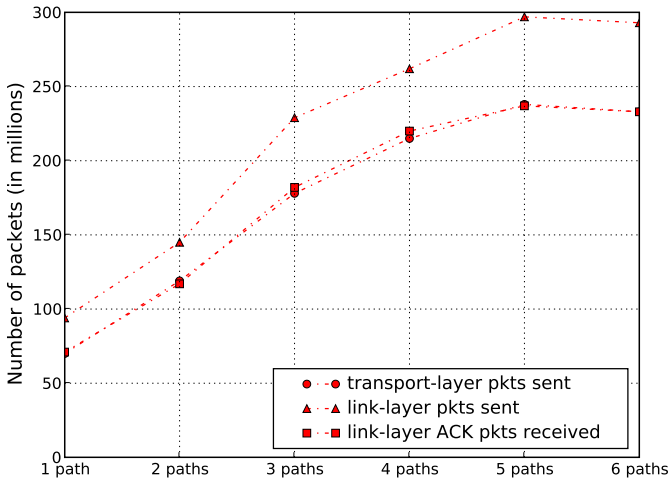


Figure 4.17: *PCIe receive performance counters. For each path and experimental run 30 million minimum-sized packets were generated.*

when bottlenecked. To do so, we used *ethtool* to retrieve general packet statistics from the network cards. As it turns out, the received packet count plus the missed packet count equals the generated packet count, and so packets are being dropped by the forwarder's receive interfaces. The Intel documentation for the e1000 driver [17] explains that packets are missed when the receive FIFO has insufficient space to store the incoming packet. One possible explanation for this is that too few buffers are allocated to cope with bursty traffic. However, increasing this number did not improve performance.

The more plausible explanation is that packets are not being DMAed fast enough. We have already shown that the CPUs have spare cycles available, and so it could not be the case that they are not polling the interface quickly enough. The next possible reason is a limitation on the PCIe bus. At first sight it would seem that this would not be the case: a card in the system can forward more minimum-sized packets when it is the only one doing so than when it is forwarding alongside the two other cards.

The PCIe statistics registers confirm this. As expected, the number of transmitted transport and link layer packets as well as that of received link layer acknowledgment packets show the typical behavior of linear increase up until the saturation point, as shown in Figure 4.17 (please note that the graph contains three curves for convenience, not for direct comparison reasons). This, combined with the fact that even at the saturation point all cards report that all transmitted packets are properly received without retries or other costly overheads (no stalls due to lack of flow control credit, no restransmitted PCIe packets, no stalls due to full retry buffers, and no negative acknowledgments in the link layer), confirms that the PCIe bus is not the bottleneck.

Memory, thus, remains as the last possible cause of the bottleneck. This makes intuitive

sense, since it and the memory controller in the north bridge are the only components that are common to all interfaces, and the bottleneck only occurs when all interfaces are used concurrently. In other work [23] we showed that the problem arises from the memory controller hub having to multiplex short transfers (the minimum-sized packets) from 12 network interfaces to and from memory, resulting in poor localisation and causing delays as memory address lines have to be continually changed.

The good news is that despite the bottleneck, cheap, off-the-shelf hardware can forward an impressive number of minimum-sized packets and can do so at line rate for most other packet sizes. Even better, future architectures such as NUMA (Non-Uniform Memory Architecture) already provide a separate memory controller and banks per CPU; careful allocation of memory to each of these could eliminate the main bottleneck discussed here, potentially resulting in important performance gains.

Chapter 5

Evaluation of Architectures

Ideally we would like the mechanisms described in the architectures to be implemented in hardware, perhaps as part of a router platform. In reality, however, and especially during the early deployment stages, it is unlikely that commercial vendors will adopt the approaches without having seen some level of real-world deployment. Consequently, we have opted to implement the solutions using off-the-shelf hardware to show their feasibility. In this evaluation chapter we will focus on the Edge-to-Edge and Terminus architectures since we believe they have the greatest potential for actual deployment on the current Internet. The second reason is that the Routing and Tunneling architecture is more suited for a hardware-based platform: at least during initial deployment the filtering elements are likely to have to handle large (carrier-grade) rates, and so an implementation based on commodity hardware might not be sufficient to show feasibility. Despite this, a lot of the filtering performance results we will present in this chapter are relevant to this architecture and might be useful when considering deployment in an ISP handling smaller volumes of traffic.

As with any performance evaluation, repeatability is key, and so using a controlled network testbed is a logical step. However, creating realistic attack scenarios in a testbed is problematic, since whatever scenarios are chosen, they could never be general enough to reflect real-world diversity. Despite this, it is possible to test each of the components of the architectures individually to see how they behave under heavy load and pathological cases, and thus provide some level of confidence with regards to the architectures' overall performance. In the rest of this chapter we present performance figures for the various components of the Edge-to-Edge and Terminus architectures; we leave discussion of these results to chapter 6. Unless otherwise stated, throughout this chapter assume that the platform for generators and counters was a Dell 1950 (recall figure 4.1), that the platform for all other components was a Dell 2950 and that the topology used was that shown in figure 4.13.

5.1 Edge-to-Edge Architecture

The two main components of this architecture are the *encapsulator* which takes care of ingress filtering, filtering malicious traffic and encapsulation and the *decapsulator*, which removes the outer IP header and can also determine which encapsulators packets. In this section we provide a detailed performance evaluation of both of these.

5.1.1 Encapsulator

The encapsulator is installed near traffic sources and is in charge of performing four functions: ingress filtering, filtering of malicious traffic, decapsulator look-up and IP-in-IP encapsulation. The first of these, ingress filtering, ensures that no packets with spoofed IP source addresses enter the network. In order to be flexible we wanted to support different mask lengths for the actual filters, but, at the same time, without taking too great a performance hit.

To accomplish this we built a custom click element called `IngressRadixFilter`. As the name implies, this element is based on a radix trie data structure, whereby each edge of the trie constitutes a number of bits of an IP address. In our case we decided to use a 7-level trie (not counting the root node as a level), the first level accounting for 8 bits of the address while all the other ones for 4. This allows us to support the most common masks (/8, /12, /16, /20, /24, /30 and /32) while still having a relatively shallow trie for performance reasons.

To illustrate, figure 5.1 shows an example of such a trie. Each of the black nodes denote where an entry would be inserted for IP address 192.168.5.100 for any of the mask lengths mentioned above. For instance, 192.168.5.0/24 would be inserted at level 5, at the node labeled 5, while 192.168.5.100/32 at level 7 node 4. With this in place, the `IngressRadixFilter` element tries to match the IP source address of a packet to the entries in its radix trie, dropping it if no match occurs.

The second function performed by the encapsulator is the actual filtering of malicious packets. For this we also implented a custom Click element called `EcapFilter` which is based on a radix trie. The difference between this and the previous element is that the latter stores simple boolean values (essentially stating whether a packet should pass or be filtered), while `EcapFilter` stores destination IP addresses, allowing victims to request filters of the form `<src IP/mask, dst IP>`. Further, the special destination address 0.0.0.0 acts as a wildcard, blocking all traffic from a certain source address, a feature perhaps useful to the local administrator if he deems a source to be malicious. A more expressive filter consisting of a destination IP address *and* a mask is of course possible by changing the data structure used to hold entries to a radix trie.

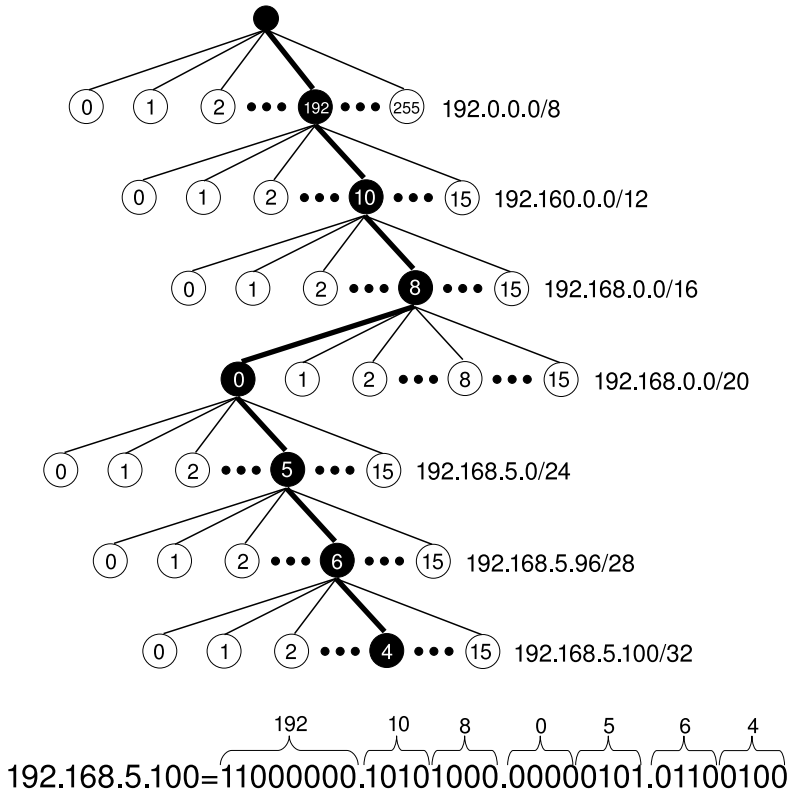


Figure 5.1: *Seven-level radix trie showing insertion of IP address 192.168.5.100 along with all its possible masks. The first trie level consists of 8 bits, all the rest 4 bits.*

The last two functions of the encapsulator are combined into the `DecapLookup` element. This element holds a list of mappings of the form:

`decapsulator IP address` → `IP address/mask`

These are the signed bindings sent over the peer-to-peer routing distribution network described in section 3.2.8, and basically describe which decapsulators are responsible for which address ranges. The mappings are stored in radix which looks up a matching prefix for a given destination and returns its corresponding decapsulator's IP address.

Upon receiving a packet, the `DecapLookup` element tries to find a matching mapping. If it does not find one, it means that the packet's destination resides at a legacy ISP and simply forwards it without further processing. If, on the other hand, a mapping matched, the element encapsulates the packet with an IP header, setting the new header's IP source address to the encapsulator's address and the IP destination address to the decapsulator's address obtained from the mapping. Finally, the element clears the evil bit on the outer header, recalculates the checksum and forwards the packet. Clearing the evil bit is done merely to show what the performance of the operation is; in an actual deployment this operation would be done by edge

routers, as described in the previous chapter.

Putting all of these elements together results in the Click configuration for the encapsulator shown in figure 5.2. The figure represents a single forwarding path, and so the full configuration file has six paths plus a `StaticThreadSched` element used to assign each path to a separate thread (and consequently one CPU core).

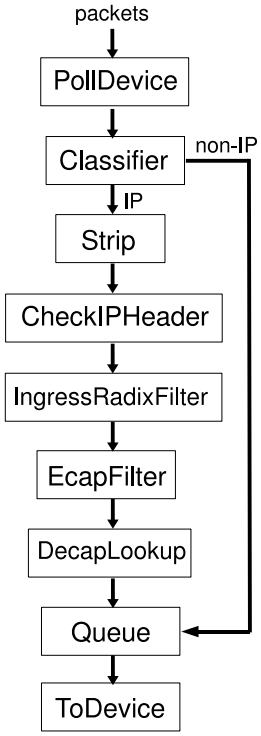


Figure 5.2: *Encapsulator forwarding path with custom Click elements.*

To conduct the encapsulator performance tests we used this configuration along with the topology in figure 4.13. The generators sent packets of varying sizes at line rate, the encapsulator forwarded them and the counters acted as sinks.

In order to make the experiment as realistic as possible we took three measures. First, we inserted 10,000 random /24 prefixes into the `IngressRadixFilter`. This is equal to about 2.5 million addresses, a number that should be large enough to cover almost all deployments. We also clearly had to insert the filters that would allow the traffic to go through so that the counters could receive it. Again, in the interest of causing the worst-case scenario we inserted a /32 filter (per path), thus forcing the encapsulator to traverse all levels of the radix trie when performing ingress filtering.

For the second measure we inserted 1,000,000 filters into the element `EcapFilter`. Considering that encapsulators reside near the sources of traffic and that the largest botnets currently being reported are in the order of 1.5 million bots, this number should be ample

enough to be able to filter even large DDoS attacks. This assumes that a very large attack does not originate entirely from sources behind a single encapsulator, a highly unlikely scenario. For this element we also inserted filters with /32 prefixes where the source IP address matched that of the generated packets but whose destination IP address did not match, thus causing packets to traverse all levels of the trie but miss. For the final measure we inserted 20,000 mappings into the `DecapLookup` element, representing the number of ASes currently on the Internet.

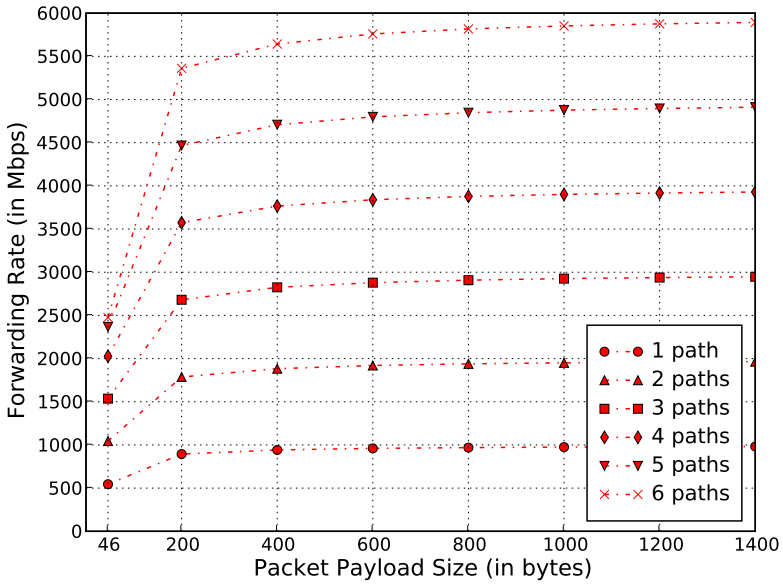


Figure 5.3: *Encapsulator forwarding performance. The encapsulator contains 10,000 ingress filters, 1,000,000 anti-DoS filters and 20,000 decapsulator-to-prefix mappings.*

With all of these filters and mappings in place we generated packets of varying sizes and for different numbers of forwarding paths, arriving at the results in figure 5.3. The graph shows that the encapsulator performs excellently for most packet sizes, forwarding packets at line rate. It further illustrates the fact that its performance scales well with the number of interfaces and CPU cores.

Minimum-sized packets present a somewhat different story (see figure 5.4). As shown, the forwarding rate as a percentage of the theoretical maximum begins at 72% in the case of one forwarding path and holds more or less steady up to four paths, demonstrating that up to this point performance scales more or less linearly with the number of paths. This pattern is similar to that of the baseline results presented in figure 4.9(b), albeit with lower numbers because of the added work done by the encapsulator.

Adding a fifth path yields only a slight increase in the forwarding rate which results in a drop in the percentage of theoretical maximum line, while adding a sixth path results in no improvement and a sharper drop. Again, this is consistent with the baseline measurements:

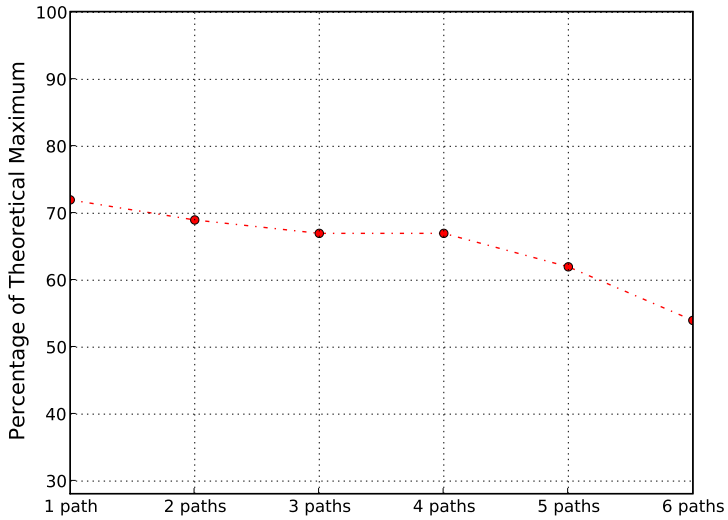


Figure 5.4: *Encapsulator forwarding performance for minimum-sized packets as a percentage of the theoretical maximum.*

using these additional paths causes the encapsulator's performance to be limited by the memory bottleneck described in the previous chapter. Despite this, the encapsulator can forward even minimum-sized packets at a very reasonable 4.8 million packets per second, or about 2.5 Gb/s.

In order to have line-rate generators, the packets used for the experiments described so far have all had the same source IP address per forwarding path. However, this can potentially skew the performance results, since this field is used to do look-ups in data structures, a process that can benefit from good cache locality. In addition, this is clearly an unrealistic scenario: traffic from clients will normally be mixed at an encapsulator and one has to assume that an attacker could try to spoof its IP address in order to degrade the encapsulator's performance.

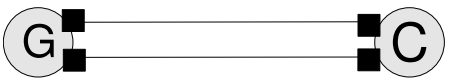


Figure 5.5: *Simple topology to test a random generator's performance.*

As a result, we wanted to see how having different source IP addresses would affect performance. Lacking a hardware traffic generator we decided to create a new Click element called `FastRandomSrc`, which, as the name implies, generates packets as fast as possible each with a different IP source address. In order to test the generation rate we used the topology shown in figure 5.5, resulting in a rate of about 525,000 packets per second per interface for minimum-sized packets; clearly, generating packets with random IP addresses incurrs costs (recall that the line-rate for minimum-sized packets is about 1.48 million pps).

We then tested the border patrol's forwarding performance with these sources. Before we

did so, we tweaked the `IngressRadixFilter` element, inserting all possible /8 prefixes so that all packets would be forwarded. Under this setup, and using the same number of filters as in the previous experiment, the border patrol was able to forward at a rate of almost 3.2 million packets per second, equal to the maximum rate sent by the generators. This result shows that the border patrol can keep up with large rates of packets where *each* has a different source IP address, and could probably do so at even higher rates had we had more powerful traffic generators.

5.1.2 Decapsulator

The decapsulator takes care of terminating the tunnel created by the various encapsulators. In addition, it is in charge of determining which of these a particular source of traffic actually flowed through. To accomplish this we created two Click elements, `TunnelFilter` and `DecapMonitor`, respectively.

The first of these elements, `TunnelFilter`, begins by looking at whether the decapsulator is the intended destination for a packet. A negative result signifies that the packet came from a legacy ISP, and so the decapsulator forwards it without further processing. At first sight letting legacy packets through like this might seem counter-intuitive. However, these packets will have their evil bit set, causing the border router to give them lower priority. If a packet is not from a legacy ISP, `TunnelFilter` next checks whether it is an IP-in-IP packet and drops if it is not, since all packets arriving at a decapsulator from an encapsulator should be tunneled. Packets that pass these tests are given to the next element in the configuration.

The `DecapMonitor` element takes care of removing a packet's outer IP header and keeping track of which encapsulator it came through. In order to achieve the latter task, it stores so-called *monitors* in a hash data structure with separate chaining. A monitor is a simple mapping between a source IP address and an encapsulator's IP address, necessary information when sending a filtering request. While it would of course be possible for `DecapMonitor` to keep such an entry for every packet with a different source IP address, this might cause a lot of unnecessary state to be kept since not all packets need filtering.

A better strategy is for the Intrusion Detection System (or whatever service is in charge of detecting attacks and communicating this information to the decapsulator) to ask the decapsulator to keep track of malicious sources, essentially installing a monitor. When the next packet from a malicious source arrives at the `DecapMonitor` element it uses the source IP address to derive the key into the hash, checking to see if a monitor exists for that traffic source. If it does, `DecapMonitor` extracts the outer header's source IP address (the encapsulator's address) and uses this to complete the mapping in the monitor. Once this is done the decapsulator could take

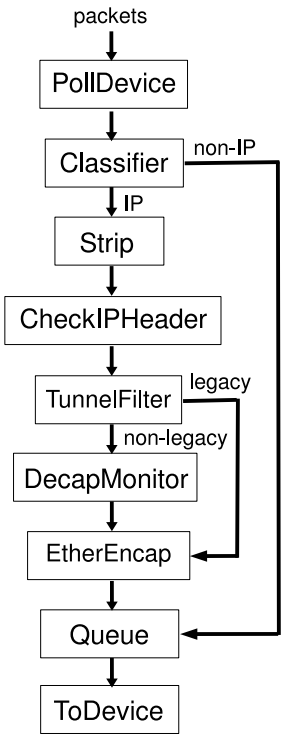


Figure 5.6: *Decapsulator forwarding path with custom Click elements.*

care of issuing a filtering request, or let some other service take care of retrieving the mapping and sending the request. Figure 5.6 shows a diagram of the full Click configuration forwarding path for the decapsulator.

Before conducting decapsulator performance tests we need a source of IP-in-IP packets. While we could have used the regular traffic generators we have been using so far in combination with an encapsulator, we did not want the latter's performance to affect the results of the decapsulator's test. As a result, we constructed a new Click element called `FastTunnelSrc` that sends tunneled packets at high rates while letting the user set the IP addresses of both the inner and outer headers.

With this element in place we then conducted a quick test to see its generation rate. Since the aim was to use the same configuration as in previous experiments, we used the simple topology in figure 5.5 to ensure that the generator could send packets simultaneously out of two interfaces at line rate. It is worth pointing out that because of the extra IP header the total size of the minimum-sized packet is now 66 bytes: 14 bytes from the ethernet header, 40 from the two IP headers, 8 from the UDP header and 4 from the CRC (see figure 5.7); the payload is 48 bytes in size. The results confirmed that the generator can send packets at line rate even for these minimum-sized packets.

For the first tests we had the decapsulator process traffic with no monitors installed to get

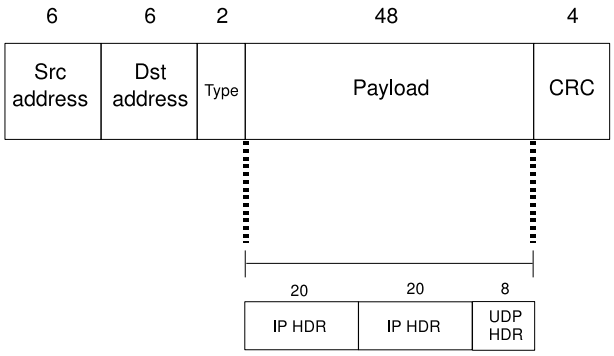


Figure 5.7: *Mimimum-sized IP-in-IP packet with UDP header with total size of 66 bytes and payload of 48 bytes. All field sizes are in bytes.*

an idea of its baseline performance. As shown in figure 5.8, in this simple case the decapsulator performs well, stripping outer headers and checking for matching monitors at line rate for most packet sizes. As with the encapsulator, performance clearly scales with the number of forwarding paths. Performance for minimum-sized packets is also as expected, scaling well up to four forwarding paths before hitting the memory bottleneck.

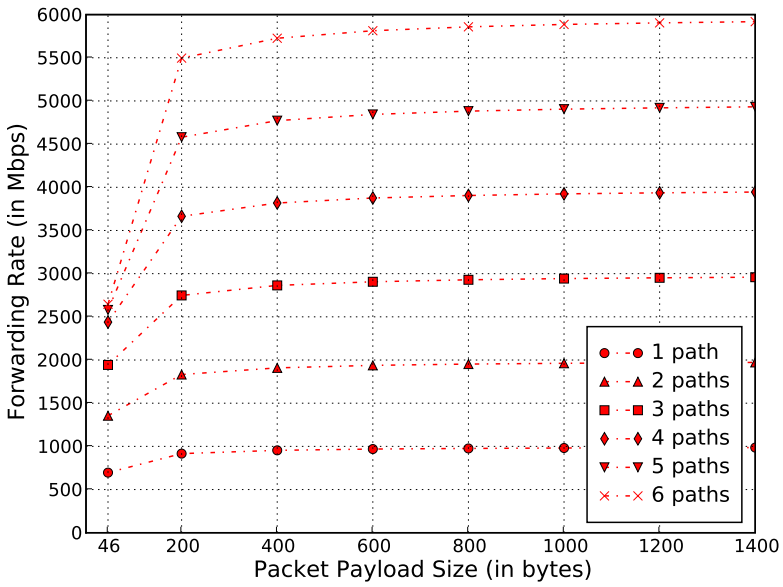


Figure 5.8: *Decapsulator forwarding performance for different packet sizes.*

How is the performance affected when monitors are installed in the `DecapMonitor` element? As mentioned, the element uses a hash with separate chaining to store monitors. In the worst case, packets would cause the element to have to traverse long chains searching for a match. To test this, we could instrument the hash function to always return the same value, thus forcing all monitors to end up in the same chain and all packets to traverse it, essentially

degrading the hash to a linked list. However, this would result in good CPU cache locality (at least for chains of reasonable length), skewing the results in our favor.

In the previous section we got around this by relying on special packet generators that sent packets with random IP source addresses. However, these could only generate packets at considerably less than line rate. Since the decapsulator's performance is largely dominated by the `DecapMonitor` element, we can tweak the hash to achieve thrashing of the CPU cache while retaining the regular, line-rate generators. Doing so is simple: change the hash so that each incoming packet hashes to a different bucket. The problem with this is that the chains can have different lengths, and since packets are hashing to different buckets, the performance results will not be a measurement of any particular chain length. To solve this we once again instrumented the hash so that all buckets in the hash had chains of equal length.

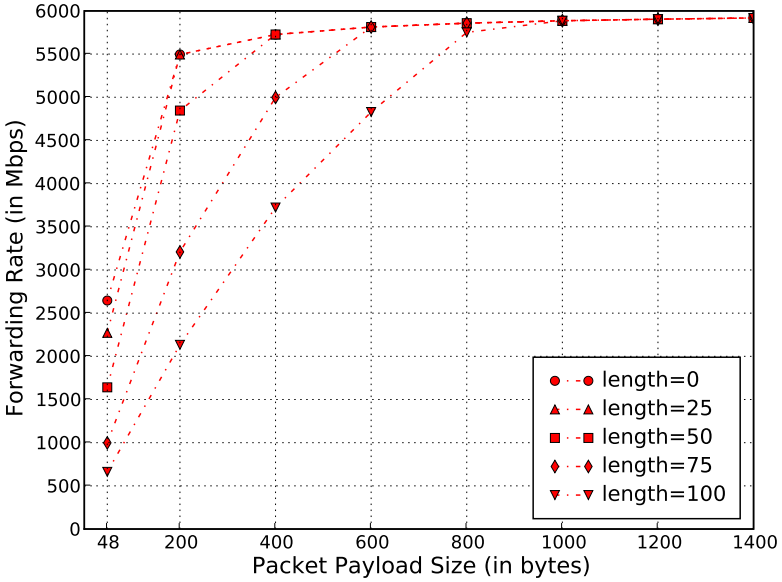


Figure 5.9: *Decapsulator forwarding performance for six forwarding paths under thrashing and hash chains of different lengths.*

With this in place we measured the decapsulator's forwarding rates for varying chain lengths and six forwarding paths (see figure 5.9). The graph shows that, even in this extreme scenario where every packet hashes to a different bucket, the decapsulator can forward at or close to line rate for payload sizes bigger than 600 bytes for very long chains of 100 nodes. For chains of length 25 the decapsulator can forward almost at line rate even for payload sizes of 200 bytes. For minimum-sized packets and this chain length the rate is lower, but does not significantly drop from the baseline figure given in the previous graph (the forwarding rate drops from 57% of the theoretical maximum to 49%). In sum, even in the case where

- minimum-sized packets are used

- each packet hashes to a different bucket causing CPU cache thrashing
- each packet forces a traversal of fairly long chains of 25 nodes
- six forwarding paths are in use

the decapsulator is still able to forward about 4.3 million packets per second, or about 2.2 Gb/s. While the `DecapMonitor` element has the ability to decrease chain length by rehashing, these results show that this operation should not be needed very often, assuming a reasonable number of buckets is chosen.

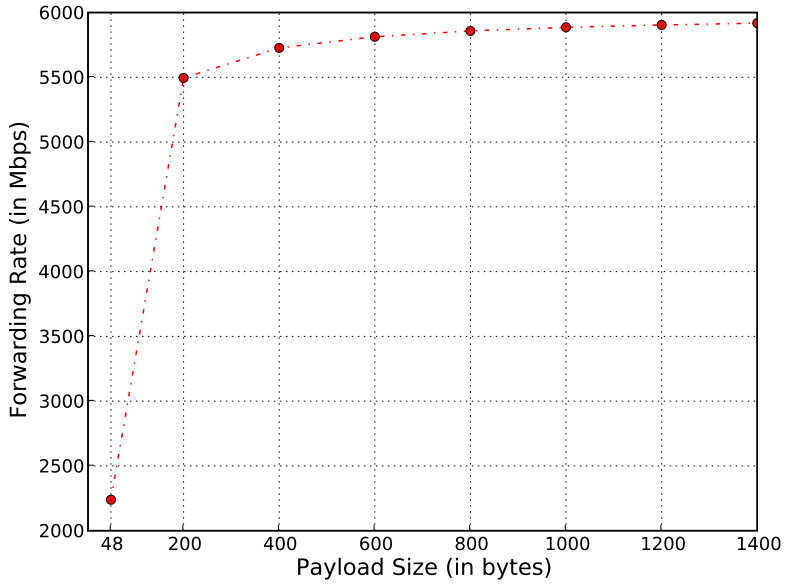


Figure 5.10: *Decapsulator forwarding performance with one million monitors installed. The test used minimum-sized packet and six forwarding paths.*

To demonstrate the decapsulator's performance under a more realistic scenario, we restored the hash's insertion routine back to normal while retaining the tweak to its look-up routine to cause thrashing. For this test we used six forwarding paths, 100,000 buckets and one million monitors, resulting in a longest chain of length 13 (the same monitors were inserted into all forwarding paths in order to achieve equal results). Figure 5.10 shows that the decapsulator can handle a very large number of monitors while still forwarding at high rates even for minimum-sized packets.

While it might seem that one million monitors might not be sufficient considering that a botnet nowadays may contain as many as 1.5 million bots or more, it is worth pointing out that one million is the number of *simultaneous* monitors that the decapsulator handled in the test. Once a monitor is filled with an encapsulator's IP address and the information retrieved it can

be erased, leaving room for other monitors to be installed. As a result, the decapsulator should be able to handle even the largest botnets.

5.2 Terminus

The Terminus architecture has three main components: the filter manager (FM) in charge of receiving reports from the IDS and generating filtering requests, the border manager (BM) which takes care of receiving and validating filtering requests, and the border patrol (BP) which filters the actual traffic. In this section we evaluate the performance of each of these in turn, first looking at the BP's forwarding plane performance, then the FM, BM and BP's control plane performance, and finally measuring the performance when these two planes are run concurrently at the BP.

5.2.1 Forwarding Plane

The forwarding plane of the Terminus architecture is essentially the border patrol. Similar to the encapsulator, the BP has an element performing ingress filtering and another one for filtering malicious traffic (in this case called `BPFilter`, see figure 5.11). However, the BP has added functionality to protect against reflection attacks based on a conditional filter of the form:

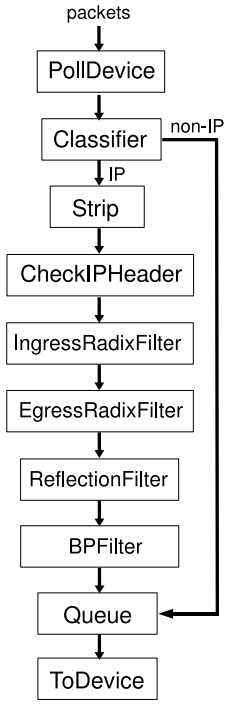


Figure 5.11: *Border patrol forwarding path with custom Click elements.*

if filter request packet arrives with $TS = 1$ **then**

At EP, block traffic from S to R where $TS = 0$

else

At BP, set $TS = 0$ on traffic from R to S

end if

While the egress patrol (EP) is logically separate from the BP, our border patrol's implementation combines the two by including two new elements: `EgressRadixFilter` takes care of the "if" clause of the conditional filter, blocking spoofed traffic from S to R if the true source bit is clear, while `ReflectionFilter` manages the "else" clause, demoting traffic from R to S by clearing its true source bit (refer to section 3.3.3.4).

To conduct the border patrol's performance tests we inserted 10,000 /24 ingress prefixes, one million border patrol filters, 500,000 egress filters and 500,000 reflection filters (corresponding to the "if" and "else" clauses of conditional filters, respectively). As usual, we also inserted ingress filters that allowed the packets to actually be forwarded. Figure 5.12 shows that the border patrol has very good performance even for minimum-sized packets.

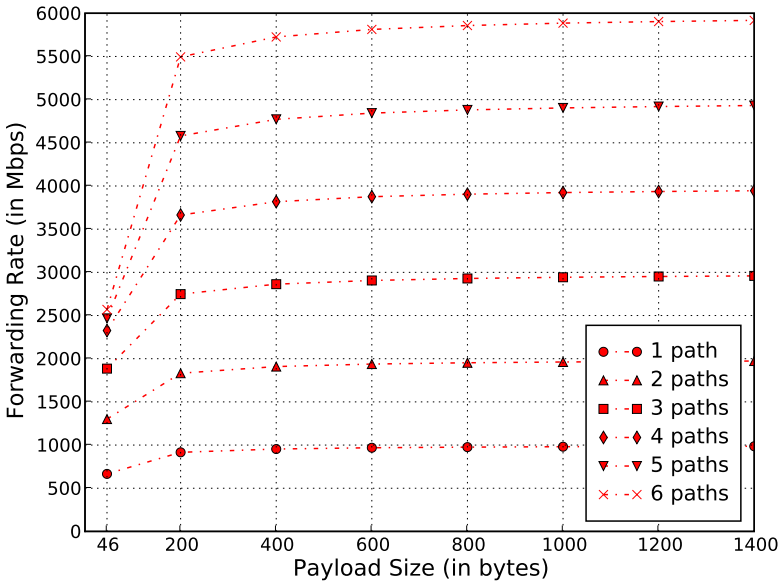


Figure 5.12: *Border patrol forwarding performance for different packet sizes.*

As a final test we conducted the same experiment as with the encapsulator, using the random generators to see if having packets with different IP source addresses degraded performance. The results were similar, with the border patrol forwarding mimimum-sized packets as fast as the generators sent them, or almost 3.2 million packets per second.

5.2.2 Control Plane

To transmit requests between the components of the control plane we implemented the Internet Filtering Protocol (IFP) described in appendix A. While the protocol supports various oper-

ations such as filter removal and retrieving attack traffic statistics, the results presented here focus on the most important operation, filter installation. To test the worst case, we used fine-granularity filters consisting of a source and destination IP address pair (though the protocol supports prefix-based and destination-only filters), and each IFP packet contained a request with a single filter. We implemented all control plane elements in C++ and used Dell 1950s for the border and filter managers and a Dell 2950 for the border patrol.

We tested the performance of each component individually. The first of these, the filter manager, listens to requests from an IDS or server. Upon receiving a filter install request, it assigns a random request number to it, looks up the mapping between the source address of the filter and the appropriate border manager, and forwards the request to that BM. When it receives a nonce from the BM, it echoes it along with the filter specification, and waits for the final installation acknowledgement from the BM (see figure 5.13). To test the FM's performance, the other components that it communicates with must not become a bottleneck. To achieve this, we implemented dummy versions of the IDS client and the BM which do the bare minimum: the IDS always sends the same filtering request packet, while the BM always provides the same nonce and immediately sends an acknowledgment upon receipt of a nonce reply, without verifying the actual nonce.

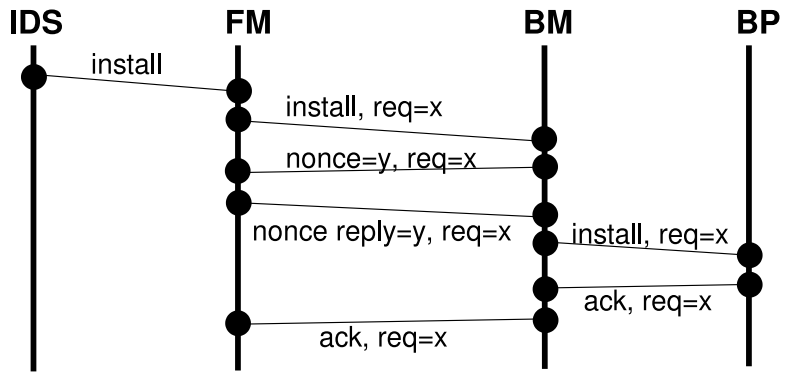


Figure 5.13: *Filter installation request using the Internet Filtering Protocol.*

With this setup, the FM was able to sustain a rate of 75,000 requests/second. To put this in perspective, the largest botnet currently reported in the media contained around 1.5 million hosts, although not all hosts in such a large botnet may be used in any attack. Even for such a large botnet and using fine-granularity src/dst IP address filters, the FM would be able to add filters for all these bots in only 20 seconds.

The second component to test in the control plane is the border manager. The BM listens to requests from FMs. Upon receiving a filter request from one of these, it sends back a nonce

which is generated from the FM's address, the request number, the filters and a secret. When it receives a nonce reply, the BM ensures that the FM has the authority to request filters for the given destination IP addresses (using mappings distributed by the peer-to-peer mechanism) and that it knows about a BP that can filter the given source address. If both of these checks succeed, the BM forwards the filter install request to the relevant BP(s), waits for an ack and forwards it to the requesting FM.

To test the performance of the border manager, we constructed dummy versions of the filter manager and the border patrol. The dummy FM always generates the same request (except that each contains its own randomly-generated request number) and sends it to the BM without looking up the mapping between the source address of the filter and the BM. The dummy BP waits for a filter install request and immediately replies with an acknowledgment for it. With this test framework the real border manager was able to sustain a rate of 87,000 requests/second. Again, this is sufficient to filter even the largest botnets in a matter of seconds.

The last control plane element is the border patrol. The BP receives filter installation requests, installs them in the filtering element of the forwarding plane, and sends and acknowledgement back to the requesting BM. Once again, we used a simplified dummy version for the BM, ensuring that the BP was the bottleneck. For this experiment the control plane and forwarding plane of the BP ran on separate CPU cores. No packets were forwarded for this experiment, since the aim was to test the performance of the control plane.

Initial testing of this setup resulted in the BP being able to handle about 54,000 requests/second. However, further testing revealed that this figure was limited by how fast we were able to write to the '/proc' directory, which is used by the filtering element in the Click router to accept filters. Indeed, this version of the BP was writing to this directory once per filter, degrading performance. To improve the figure, we modified the BP to install filters in batches. Using a 100-filter batch size, the BP was able to install filters into the forwarding plane at a rate of about 354,000 requests/second, an order of magnitude difference.

It is of course possible to increase the batch size beyond 100, but this size already results in a rate that is more than sufficient to filter any malicious sources sitting behind the BP in very little time. Further, as shown in figure 5.14, this rate does not significantly increase with larger batch sizes. It is worth mentioning that a batch size of 100 means that *at most* 100 filters can be installed with one write to the '/proc' entry: a victim needing a single filter could do so without having to wait for another 99 filters to come along.

To sum up, the filter manager, border manager and border patrol are able to handle requests at rates of 75,000, 87,000 and 345,000 requests per second respectively. While we did not

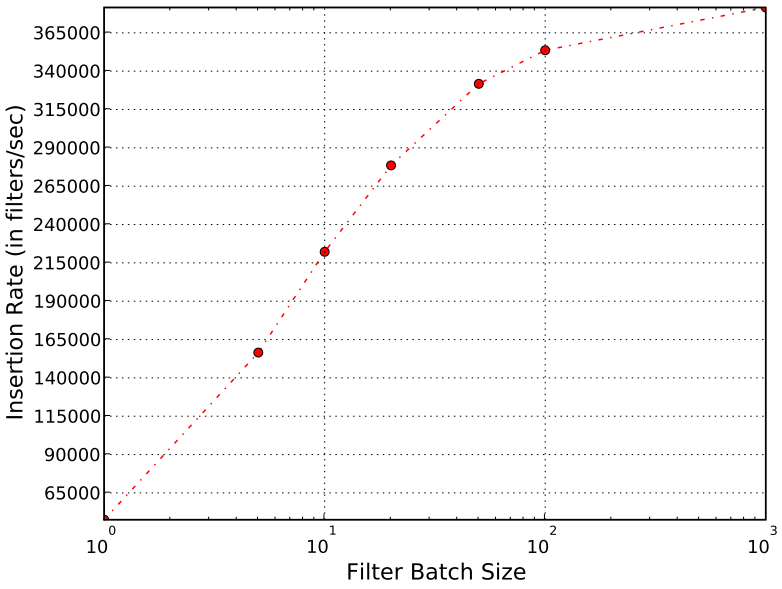


Figure 5.14: *Border patrol's filter insertion rate for different batch sizes.*

particularly optimize the performance of any of these components, these results clearly show that the control plane of the architecture would be able to filter even the largest botnets in a matter of seconds.

5.2.3 Combining the Two Planes

For the final test, we wanted to determine how running the BP's control plane would affect its forwarding performance and vice versa. To do this, we used the same control plane components as in the previous section while simultaneously forwarding minimum-sized packets on all six forwarding paths. Under this scenario the border patrol was able to forward packets at the usual rate of about 2.5 Gb/s while inserting filters at 345,000 requests per second.

This shows that the control plane has very little impact on the forwarding plane as long as they are run on separate CPU cores. Indeed, conducting the same test on a uni-processor kernel causes the forwarding rate to plummet to 78 Mb/s per path (the filter insertion rate remained the same). These figures clearly show that the border patrol can install filters at a very high rate without impacting high speed forwarding, and that modern multi-core CPUs are very well suited to this task.

Chapter 6

Discussion and Conclusions

In the previous chapters we introduced three architectures against Denial-of-Service attacks: Routing and Tunneling, Edge-to-Edge and Terminus. In addition, we presented in-depth evaluations of the baseline performance of the platforms used as well as the performance of the architectures' various components. In this chapter we begin by summarizing these results, explaining why they give us confidence that the architectures and their components can cope with even the largest botnets currently in existence. In addition, we touch upon incentives for deployment and provide a comparison between the architectures; we close the chapter with a discussion of some remaining issues.

6.1 Implementation Results

The previous chapter presented forwarding performance results for the encapsulator, decapsulator and border patrol, as well as for the control plane of the Terminus architecture. In the case of the forwarding plane, these results showed line-rate performance for almost all packet sizes except minimum-sized ones. What packet sizes are the architectures' components likely to see? Different measurement studies give different results [12; 13; 52] depending on where the monitor was located and when the study was performed. The packet size distribution appears to have strong modes at 40 bytes and 1500 bytes, with a range of values in between. Mean packet sizes appear to range from 200 to 600 bytes, which bodes well for our implementation, since we can deal with these at line-rate.

Even in the worst case, if all the traffic were 40 byte TCP Acks (essentially minimum-sized packets), we showed that we would be able to process this at rates of about 2.4, 2.2 and 2.6 Gb/s for the encapsulator, the decapsulator and the border patrol, respectively. These figures are even more impressive if we consider that an outgoing rate composed entirely of TCP Acks corresponds to a much larger incoming TCP data rate.

In addition, we showed that the architectures' components can achieve these high rates

while dealing with a large amount of state. The encapsulator and border patrol, for instance, forwarded at these rates while having as many as one million anti-DoS filters. Since these components are installed near sources of traffic and considering that currently the biggest botnets have about 1.5 million bots, one million filters should be more than sufficient to filter several very large attacks (assuming of course that these attacks do not originate entirely behind the encapsulator or border patrol, a highly unlikely event). We also showed that the border patrol could cope with a very large number of filters against reflection attacks.

The encapsulator and border patrol also had 10,000 /24 ingress prefixes installed, equal to about 2.5 million hosts. This number need not, in fact, be so high, as shown by the following back-of-the-envelope calculation. If an encapsulator or BP is sized to handle a fully-loaded 1 Gb/s upstream Ethernet link, then how many prefixes would this correspond to at a broadband ISP? With a 512 kb/s upload speed per customer and an over-subscription ratio of 20:1 (a typical figure for ADSL) we end up with about 39,000 addresses, or about 150 prefixes for small /24 prefixes. While these numbers are rough, they show that the 10,000 prefixes used in our experiments should be largely sufficient for most deployments.

The encapsulator had additional state: 20,000 entries representing the mappings between decapsulators and destination prefixes. We chose the 20,000 figure since this is the number of ASes currently on the Internet, and we would normally expect each of these to have one decapsulator address. To allow for growth, we re-ran the test with 100,000 mappings and obtained similar results.

The last bit of state was in the decapsulator, which was in charge of keeping track of which encapsulators packets were flowing through. For these tests we used one million so-called monitors. Relating this number again to the 1.5 million botnet figure shows the former to be more than enough to filter even very large attacks, specially considering that monitors would normally be deleted once a filtering request is sent.

In addition, we showed that the architectures' components perform at very reasonable rates even when traffic is being spoofed so that every packet has a different IP source address. In fact, even in this extreme scenario the encapsulator and border patrol were able to forward at a rate equal to that sent by the traffic generators (about 3.2 million packets per second for minimum-sized packets). In the case of the decapsulator we were able to tweak the hash it relies on in order to use faster generators; this resulted, once again, in very reasonable forwarding rates of about 4.2 million packets per second.

Finally, we presented performance figures for the control plane of the Terminus architecture. We showed that even the slowest components could process filtering requests at a rate of

75,000 per second. Even in the case where every request contained a single filter, it would take only 20 seconds to filter 1.5 million malicious sources. In sum, the performance results we presented give us confidence that the components and architectures should be more than adequate to cope with very large attacks in most deployment scenarios.

6.2 Deployment Incentives

Under full deployment, the architectures would clearly provide significant protection against DoS attacks for all hosts. However, this "common good" argument is not enough on its own to motivate early adopters, since entities on the Internet generally act only in their own self-interest. To bring about change, a solution must provide incentives even for those early adopters, or it will never see any important level of deployment. In the rest of this section we will discuss the deployment incentives that ISPs have for each of the three architectures.

6.2.1 Destination ISPs

ISPs hosting potential victims have the clearest incentive, since they can charge for the protection they provide. Alternatively, such an ISP could provide this protection for free, attracting customers from ISPs that do not provide this service. Deployment for the Edge-to-Edge architecture is as simple as installing a decapsulator, including a mapping between it and the prefixes it is responsible for into the peer-to-peer network, and installing a diffserv rule at edge routers to give lower priority to packets with their evil bit set (we assume the IDS to be already in place).

The routing and tunneling architecture's deployment is more involved, but still reasonable. The ISP begins by installing encapsulators at the edges of its network, a function that could possibly be done by border routers. Further, the ISP deploys a decapsulator responsible for protecting potential victims. Finally, routes are added so that any traffic intended for these protected destinations is forced to traverse an encapsulator, and, in turn, the decapsulator.

In the case of the Terminus architecture the process is again simple: set up a box to act as the filter manager, configure it to receive the source IP prefix to BM mappings from the peer-to-peer network, obtain a certificate to sign its prefix-to-FM mapping, and install a diffserv rule at the edge routers so that packets with their true source bit set receive higher priority.

Although not required, we would also expect ISPs to deploy encapsulators for the Edge-to-Edge architecture and border managers/border patrols for Terminus, thus accelerating the acceptance rate of the architectures.

6.2.2 Source ISPs

A source ISP has less incentive for deployment, since it is not directly affected by attacks. However, in the case of the Edge-to-Edge and Terminus architectures, customers wanting to connect

to the protected servers will pressure their ISPs to deploy encapsulators or border patrols. Having a legacy provider will mean that a client receives lower priority than one connected through a non-legacy ISP. Not only may delays be longer during normal operation, but also little or no service will be given if the server is attacked. However, we are not convinced that such customer pressure will be sufficient, especially in the early stages of deployment, to entice ISPs to deploy additional equipment. The costs may be small, but they are there nonetheless. Unfortunately, the routing and tunneling architecture does not have this incentive, since there is no evil/true source bit.

Another incentive for source ISPs is a potential for a reduction in tech-support costs. Encapsulators and border patrols should have no false positives - if a receiver does not want traffic, only then it is shutdown. In the absence of false positives, encapsulation filters automatically isolate the bad behavior of a compromised client host, and should require less human intervention than current, mostly manual, mechanisms for dealing with compromised hosts. The hope is that deploying an encapsulator or border patrol allows the ISP to deal with a bot being used in a DoS attack at their leisure, reducing operational costs.

The fact that the actual deployment is simple should provide one more, albeit small, incentive. In the case of the Edge-to-Edge architecture this entails setting up a fast PC to act as an encapsulator and configuring it to receive decapsulation routes from one or more peers. For Terminus, deployment consists of installing a border manager and a border patrol, setting the BM to receive the mappings between destination address and filter managers, and obtaining a certificate to sign its prefix-to-BM mapping.

6.2.3 Transit ISPs

Transit ISPs have the weakest incentive, but may be persuaded to deploy by a client ISP (either a source or destination ISP) that has deployed one of the architectures. The transit ISP's reputation might also motivate it to implement the scheme. Fortunately the changes needed are minimal for the Edge-to-Edge and Terminus architectures and require no additional hardware: just configure border routers to set or unset the diffserv codepoint corresponding to the evil or true source bit. The Routing and Tunneling architecture is slightly more complicated, requiring changes to the behaviour of the BGP border routers as described in section 3.1.4.

6.2.4 Initial Deployment

It is easy to see that with these architectures incentives are such that increasing deployment increases the incentive to deploy. The question then is how to persuade potential early adopters to deploy.

Any site that is attacked has incentives to deploy decapsulators or filter managers, but only if they allow the attack to be reduced. However this requires that sufficient traffic traverses an encapsulator or border patrol, and there is little incentive for that to happen until many decapsulators/filter managers are deployed.

One option would be for a large ISP or Internet Exchange Point to offer an encapsulation or border patrol service, much like a source ISP would do. The site under attack could then subscribe to this service and all their incoming and potentially malicious traffic would be routed there to be processed by an encapsulator or border patrol. This would bootstrap the deployment of decapsulators or filter managers and then, sometime later, there would be incentive to deploy encapsulators/border patrols directly at customer-facing ISPs.

6.3 Comparison of Architectures

While the Routing and Tunneling architecture would certainly be effective against very large DoS attacks under full deployment, it does come with shortcomings. The first of these is that the distribution of routes relies on BGP, putting a burden on global routing, and generally requiring de-aggregation of these routes. Further, the marking and filtering of packets happens fairly close to the destination, so that each control point has to be able to handle potentially large amounts of traffic. Finally, the architecture has no anti-spoofing mechanism. Despite these problems, the architecture may still be a viable option for larger ISPs with many edge routers, so that the incoming attack can be diffused across several of these control points.

The Edge-to-Edge and Terminus architectures provide several advantages over the previous one. For one, they rely on an independent and robust peer-to-peer system for route distribution, and so do not put extra burden on BGP. In addition, their mechanisms are much simpler and provide better incentives for deployment. These architectures also provide important anti-spoofing mechanisms and better deployment incentives than the the Routing and Tunneling solution.

Terminus has a slight advantage in that it does not rely on an extra IP header to determine the sources of traffic, and so it is more efficient. However, it does need extra information distributed via the peer-to-peer network to indicate the mapping of border patrols to sources of traffic. In the end, this boils down to a trade-off between doing more work in the forwarding plane or the control plane. In general it is better not to burden the former, since it has to process packets at high rates.

Having said that, the Edge-to-Edge architecture also has an advantage over Terminus. Consider figure 6.1. Under the Terminus architecture, the fact that attacker A resides in a legacy

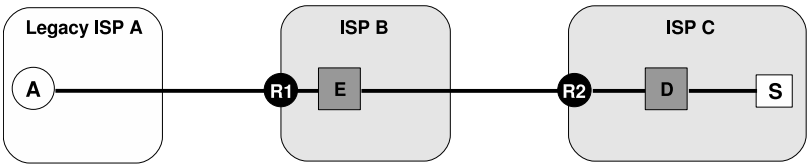


Figure 6.1: *Edge-to-Edge architecture scenario with encapsulation and filtering being done by an intermediate ISP. E stands for encapsulator, D for decapsulator, S for server and R for router.*

ISP means that R1 would clear the true source bit, and so ISP C would have no way of knowing whether the IP source address for attack packets is being spoofed or not. As described so far, the Edge-to-Edge architecture would exhibit similar behavior. However, it is entirely possible for an intermediate ISP such as ISP B in the figure to encapsulate packets coming from legacy ISPs, clearing their evil bit in the process. As a result, while attacker A is hosted at a legacy ISP, ISP C can still protect its server S by sending filter requests to encapsulator E, something that the Terminus architecture cannot do.

As shown, both architectures have their advantages and disadvantages. In the end, it is important to note that both of these should be very effective at quickly filtering very large DoS attacks, as evidenced by the performance results presented in the previous chapter.

6.4 Final Issues

In this final section we will cover some remaining issues regarding the architectures. Since, as explained in the previous sections, the Routing and Tunneling architecture has important shortcomings, we will focus mostly on the Edge-to-Edge and Terminus architectures.

6.4.1 State Attacks

The encapsulators, decapsulators and border patrols keep different state in order to perform their functions. Could an attacker insert a significant amount of state in the hope of leaving no room for legitimate state or degrading the components' performance? In the case of the decapsulator, the state consists of the monitors used to figure out which encapsulators sources of traffic are flowing through. These monitors are installed by the IDS, leaving no vector for attack; we could of course assume the IDS to be malicious, but in this case it could cause more serious damage by simply not reporting any attacks.

The encapsulator and border patrol share two types of state: ingress prefixes and the actual filters. The first of these is installed by the network manager and so is safe from state attacks. The filters (including the reflection attack filters in the case of the border patrol), on the other hand, are installed as a result of filtering requests. An attacker could create unnecessary state by sending spurious filtering requests. It is important to note that this would not be easy, since the

sources of requests are checked to ensure that they are legitimate decapsulators or filter managers. Assuming that the attacker was able to compromise such a source, the encapsulator or border patrol (or in the case of Terminus even the border manager) can keep track of the number of requests sent by each decapsulator or filter manager. This constitutes a small amount of state which could be use to rate-limit the requests. In addition, the requests could be timestamped in order to expire old entries. The encapsulator has one last type of state: the mappings distributed over the peer-to-peer network. These are digitally signed, so not susceptible to attack.

6.4.2 Link Flooding

The second issue has to do with actually sending the filtering requests. If the destination ISP's link is completely flooded, could this prevent requests from being sent out? Generally the downlink and uplink are separate, and so the only way this could happen is if the uplink were flooded; this could be the case if the server under attack was generating sufficient responses to the attack and the legitimate traffic. In this case the ISP could momentarily shape this traffic to free up some of the uplink's bandwidth, letting the filtering requests to go through. Once the filters are installed and the attack subsides the ISP can resume normal operations on the uplink.

6.4.3 Compromised Components

What would be the effect of a compromised component on the architectures? We already discussed how to protect encapsulators and border patrols against compromised decapsulators and filter managers. Of course their compromise would also result in potentially no protection from attacks, but this is a local problem: the compromised components and the protected servers are all in the same ISP which has, as a result, a clear incentive to deal with the problem.

A compromised IDS could cause generation of spurious filtering requests. Again, the border patrol or encapsulator could use the mechanisms described previously to prevent this, or a similar solution could be installed at the filter manager. Even so, it should be noted that a compromised IDS can probably cause more damage by simply not reporting attacks, a problem outside of the scope of the architectures.

What about the encapsulators or border patrols? In the worst case, these could spoof the sources of traffic, potentially causing decapsulators or filter managers to issue filtering requests. Since the sources were spoofed, these requests will arrive at the "wrong" encapsulator or border patrol. These will have no way of knowing that the request was a result of spoofed traffic, so they will install the filters as in the normal case. This is essentially the fourth scenario presented in figure 3.8 in the Edge-to-Edge architecture, and so we suggest the same solution described there. In order to separate the legitimate traffic from the bad one the decapsulator/filter manager

provides a random number to the encapsulator/border patrol to mark subsequent packets with; such information could be contained in the outer IP header for the Edge-to-Edge architecture or in the Type-of-Service field in the case of Terminus. In this way the malicious traffic can once again be distinguished.

The final component is the border manager. If it were compromised it could ignore filtering requests or cause malicious filters to be installed. Ignoring requests would allow potential attackers behind the border patrol it is responsible for to continue sending malicious traffic unhindered. To counteract this the filter manager at the victim's site could request that filters be installed at intermediate ISPs such as ISP B in figure 6.1[1]. Even if this were not possible, at the very least the victim is no worse off than if the architecture had not been deployed. In the second case, the compromised border manager could cause malicious filters to be installed. This becomes once again a local problem, and it would be up to the ISP to restore the border manager to normal so that its legitimate clients (the sources of traffic) are not blocked.

6.4.4 NATs

In general, NATs with few clients behind them do not present a significant problem. In case of attack, the victim would simply filter all traffic coming from the NAT's IP address. This could cause collateral damage if not all the clients behind the NAT were sources of malicious traffic, but not a significant amount; again, this becomes a local problem: a local and malicious host causes DoS on hosts at the same location.

What about NATs with many clients behind, as is often the case with enterprises? In this case the collateral damage from blocking all traffic from the NAT's IP address to the victim could be noticeable, specially if the victim is an important server. In a sense, this results again in a local problem with clear incentives: it is up to the administrator responsible for the NAT to find the malicious sources and remove them, causing the filter to be, in turn, removed.

While this would give the ISP managing the NAT strong incentive to remove the malicious sources, such a draconian approach might turn certain ISPs away from deploying the architectures in the first place. In essence, what is needed is to augment the filters to include source port information, thus reducing collateral damage by re-introducing the ability to single out individual hosts behind the NAT for filtering.

Introducing such a mechanism would require some changes. First, the IDS at the victim's ISP would have to report malicious behavior based not only on source IP address but also source port. Further, the filtering protocol would have to be augmented by defining a new type of filter

[1] It is worth pointing out that this is only a possibility for Terminus if the source ISP has the architecture deployed. Since in this scenario the border patrol is not compromised, we can rely on its ingress filtering functionality.

that would include source port information. In addition, the filtering box (be it an encapsulator or a border patrol) would then have to be extended to support port-based filtering. Moreover, it is worth pointing out that no changes to the architectures' routing mechanisms would be needed, since these would be based on the IP address of the NAT, not on port information. A final optional step would be to establish some form of secure signalling between the encapsulator/border patrol and the administrator of the network behind the NAT. This would serve to let the administrator know that a port-specific filter has been installed, and he could then look at the NAT's state (i.e., its mappings of ports to internal IP addresses) to pin-point the malicious host behind the NAT and remove it. In sum, relatively small changes to the architectures would allow them to cope with even large NAT deployments.

6.5 Future Work

Like with any other large piece of work, there are a number of directions that could be followed in order to further improve this thesis and the architectures presented in it. Most of the ones discussed in this section have to do with initial deployment issues (for the reminder of this section we will set the routing and tunneling architecture aside since it has more difficult deployment problems as explained in section 6.3). In particular, perhaps the single most important issue arises from the architectures' requirement that a packet traversing a legacy ISP receive lower priority.

The first direction regarding this problem would be to analyze it by looking at AS topology information from the Internet. The aim would be to understand how the level of deployment (i.e., the number of deployed ASes) affects the number of deployed paths on the Internet. For instance, such an analysis would be able to make statements like "with x% of the ASes deployed, y% of all paths are protected against DoS". This analysis would of course have to take into account the fact that the Intenet's AS topology is not uniform (deployment at Tier-1 ASes affects a much larger number of paths than at an edge AS), but would useful by giving an idea of how deployment of the architectures might progress.

A related analysis would be to determine how many (if any) of the ASes are purely transit ASes (i.e., ASes that do not source any traffic), which could be done by looking at BGP route information from the public Internet. If such ASes exist, ASes connected to them would not have to change the value of the evil/true source bit, helping the deployment story.

Another research possibility aimed at aimed at improving the deployment story has to do with scenarios where an AS has one of the architectures deployed, but its traffic happens to traverse a legacy AS, thus forcing it to receive lower priority at the destination (this is the case

of ISP C in figure 3.11). In this scenario we would essentially like some way of "hopping" or tunneling over the legacy ISP. A simple solution would be ask the source ISP (ISP C in the figure) to include a periodically-modified nonce in its packets. Such a measure would work under the assumption that the legacy ISP (ISP F) would not eavesdrop the nonce and generate packets based on it, a safe assumption for larger ISPs such as Tier-1s. For non-trusted legacy ISPs a simple nonce would not be enough, and a cryptographic tunnel could be used to "skip" over the legacy ISP. However, the feasibility of such approaches remains to be seen, since both the nonce and cryptographic solutions would require additional (and potentially taxing) work from the edge router of the downstream ISP (ISP G in the figure). Either way, such a solution would only be needed during initial deployment, and would be removed as deployment progressed.

6.6 Conclusion

Despite serious under-reporting, the number and size of Denial-of-Service attacks continue to rise, causing victims to incur considerable costs and threatening their livelihood. Many research solutions have been proposed over the past years to combat these attacks, but unfortunately they have presented difficult deployment hurdles and so have not seen the light of day at any significant level in the Internet. Commercial solutions also exist, but these are costly and generally do not scale to Internet-wide levels.

In this thesis we have presented three filtering architectures against large DoS attacks that aim at lowering these deployment hurdles: the Routing and Tunneling architecture, the Edge-to-Edge one and Terminus. While different, in essence they all give a receiver the power to simply ask that the network stop sending unwanted traffic.

Such a powerful mechanism must be introduced with great care, lest it become a Denial-of-Service tool in its own right. To this end, we have discussed at length possible attack vectors, showing that in the worst scenarios the hosts are no worse off than if the architecture had not been deployed at all. We accomplish this, among other things, by introducing easy-to-deploy anti-spoofing mechanisms in the Edge-to-Edge and Terminus architectures. We have further described the result of the various architectural components being compromised, again explaining how these would have only localized impact or "no worse off" behavior.

While we believe that the various components of the architectures are well suited to being implemented in fast routing hardware, in the early stages of deployment this is unlikely to be the case. Because of this, we have shown that these components can also be built using low-cost off-the-shelf hardware and open-source software. First we presented an extensive

baseline performance evaluation of the platform used to build these components, showing forwarding rates for various packet sizes and identifying an underlying memory bottleneck when forwarding large amounts of minimum-sized packets. We then conducted performance tests on our implementation of the various components, giving confidence that these can cope with the largest attacks, both in terms of filter installation rates and forwarding rates even for minimum-sized packets. We have shown that their performance scales well with the number of CPU cores and network interfaces, which bodes well for our solutions if we consider the current trend in computer design.

In addition, we spent time describing deployment incentives and initial deployment scenarios. We have discussed how the architectures, specially Terminus and the Edge-to-Edge one, provide clear incentives for ISPs that adopt them early, both at the destination as well as at the sources of attacks. The architectures are not only incrementally deployable but require no changes to end systems and only minimal configuration changes to network routers. Finally, we provided a comparison of the three architectures, mentioning their advantages and disadvantages.

While the Denial-of-Sevice attacks currently seen on the Internet are severe, there has not yet been one that could completely take out critical infrastructure. It would be in our best interest to ensure that we have an architectural solution against large DoS deployed should such an attack ever take place; this thesis is hopefully a step in that direction.

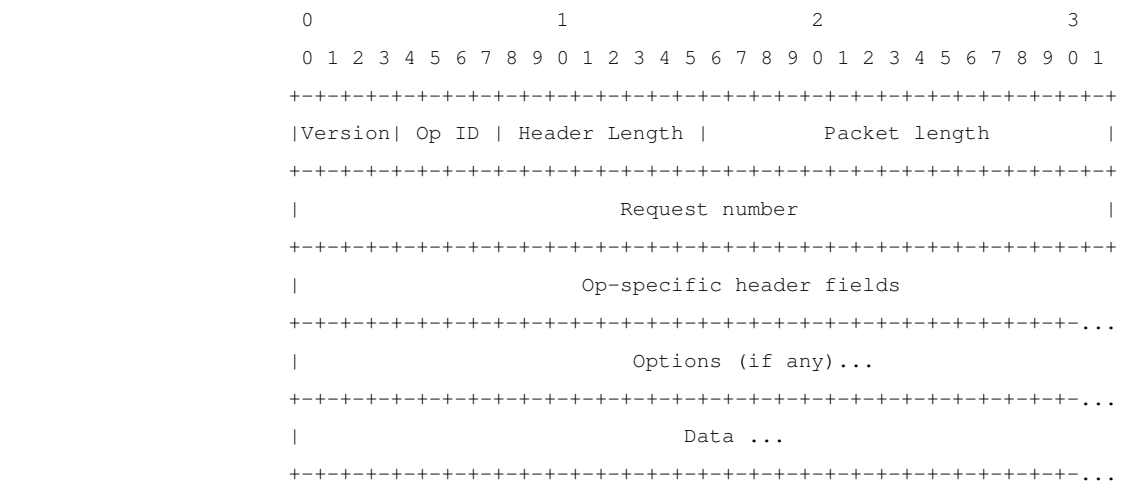
Appendix A

Internet Filtering Protocol

The Internet Filtering Protocol (IFP) provides a mechanism to communicate packet filtering information between hosts, using UDP as its underlying protocol. The protocol has two main purposes: to allow entities to request installation and removal of filters and to allow retrieval of statistics. The latter provides a general monitoring function, but can also be used by an attack victim to determine whether it is no longer receiving malicious traffic because the filters are working or because the attack itself subsided. In addition, the protol provides a simple nonce mechanism so that receivers can verify that the source of a request is not spoofed.

A.1 Common Header Format

An IFP packet consists of 8 bytes of common fields followed by a set of operation-specific fields of variable length. The text below shows the packet format as well as an explanation of the common fields; operation-specific fields are described in the next section. Any fields marked ”Reserved” or R should be set to all 0s and ignored on receipt.



Version: 4 bits

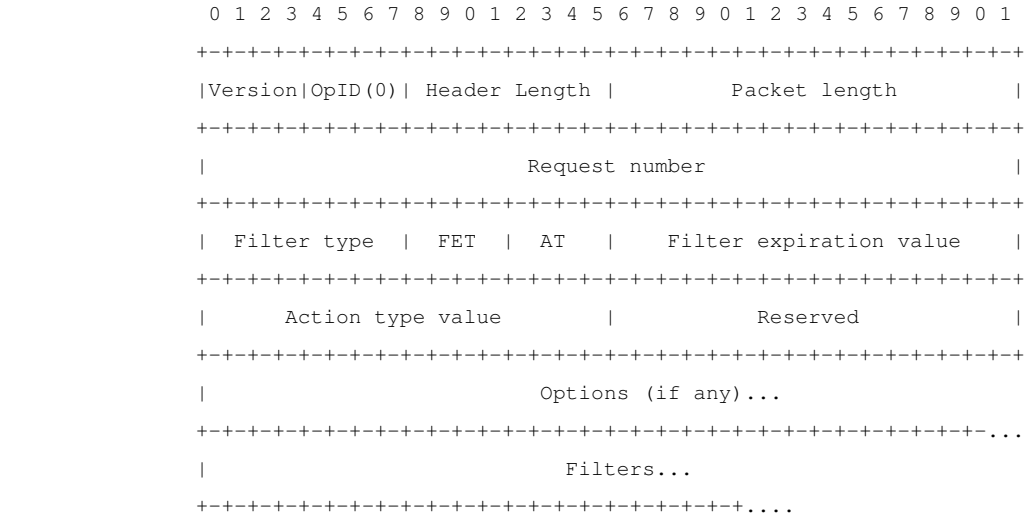
Indicates the filtering protocol’s version, starting at version 1

```
Op ID: 4 bits
   The id of the operation that the packet is requesting.

Header length: 8 bits
   Indicates the total length of the packet's header in bytes, including
   any options.

Packet length: 16 bits
   Indicates the total length of the packet in bytes.

Request number: 32 bits
   Used to associate an id with a request. Any replies to the request will
   include this number.
```

A.2 Supported Operations

This section describes in detail each of the 8 operations supported by IFP: install filters, remove filters, nonce, nonce reply, ack, error reply, statistics request and statistics reply.

A.2.1 Install Filters (op id=0)




```
FT (filter type): 8 bits
   Indicates the type of filters contained in the packet. The types of filters
   supported by version 1 of the protocol are:

      (0) Source IPv4 addresses / destination IPv4 addresses(both 32 bits).
          Each filter is 8 bytes long.
      (1) Filter all packets that the receiver of the IFP packet is in charge
          of filtering and going to the IPv4 destination in the filter. Each
          filter is 4 bytes long.
      (2) Source prefix-based. The receiver of the IFP packet has to take care of
          filtering all packets whose source match the given IPv4 prefix and going
```

to the given destination. Each filter is 9 bytes long (4 for the source IP address, 4 for the destionation IP address and 1 for the source prefix mask).

FET (filter expiration type): 4 bits
Indicates the type of timeout to apply to all filters in the packet. Supported types are:

(0) No expiration (requester much explicitly remove filter, FEV should be set to 0.
(1) Time-based expiration, in minutes. The FEV field holds the number of minutes.
(2) Expire if less than x number of packets hit the filter in the last y seconds. In this case, the first byte of the FEV field holds x, the last byte holds y.

AT (action type): 4 bits
Indicates the action to take when a packet matches the filter. Supported codes are:

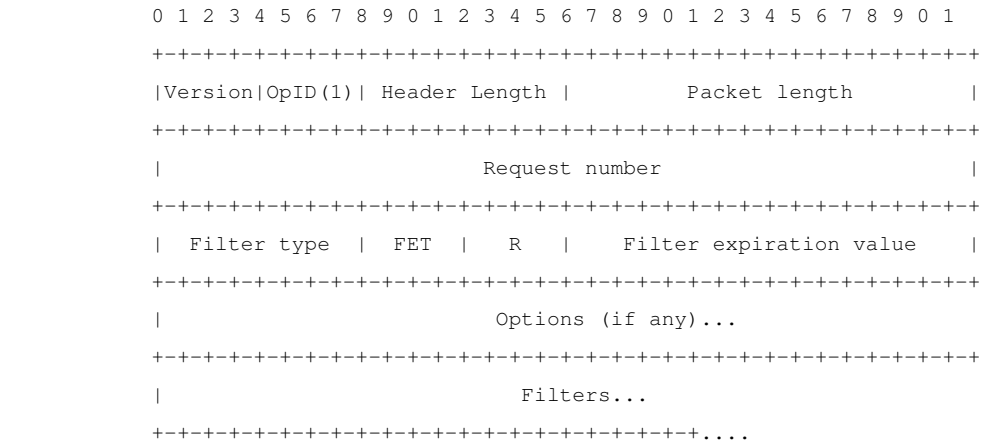
(0) Discard, drops any packets matching the filter.
(1) Rate limit by x packet count in y minutes. Any packets exceeding x in y minutes will be discarded. The first byte of the ATV field contains x, the second byte y.
(2) Rate limit by token bucket?

FEV (Filter expiration value): 16 bits
The value associated with the FET code specified (see above).

ATV (action type value): 8 bits
Contains the value associated with the AT code specified (see above).

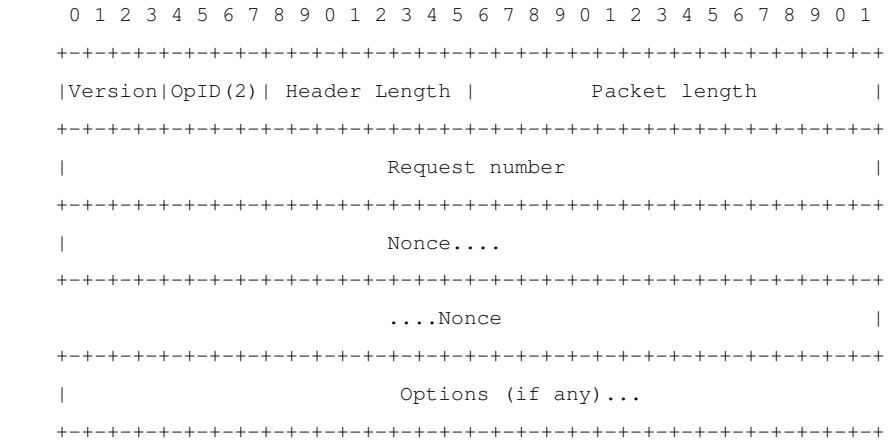
Filters: variable size
The actual filters (see FT field above).

A.2.2 Remove Filters (op id=1)



```
FT (filter type): 8 bits
   Indicates the type of filters contained in the packet (see above
   for details).

FET (filter expiration type): 4 bits
    Indicates the type of expiration to apply to filters in the packet.

    (0) Remove filters immediately, FEV should be set to 0.
    (1) Remove filters in FEV minutes.
    (2) Remove filters if less than x number of packets hit them in y seconds.
        In this case, the first byte of the FEV field contains x, the second
        byte y.

FEV (Filter expiration value): 16 bits
    The value associated with the FET code specified (see above).

Filters: variable size
   The actual filters
```

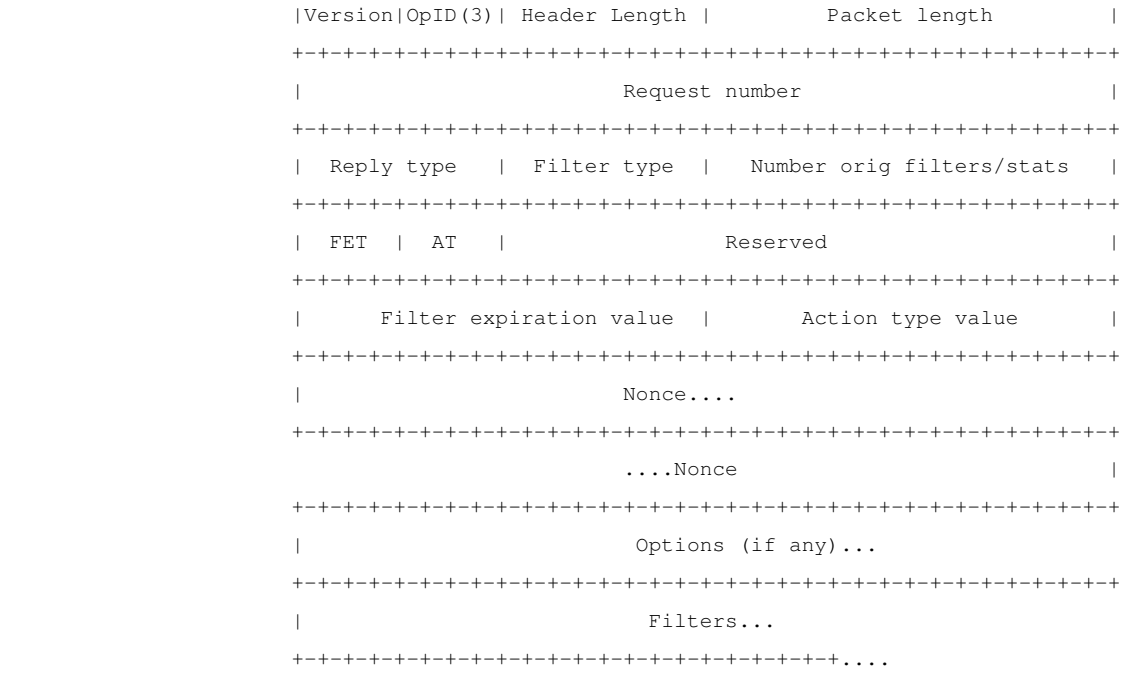
A.2.3 Nonce Request(op id=2)



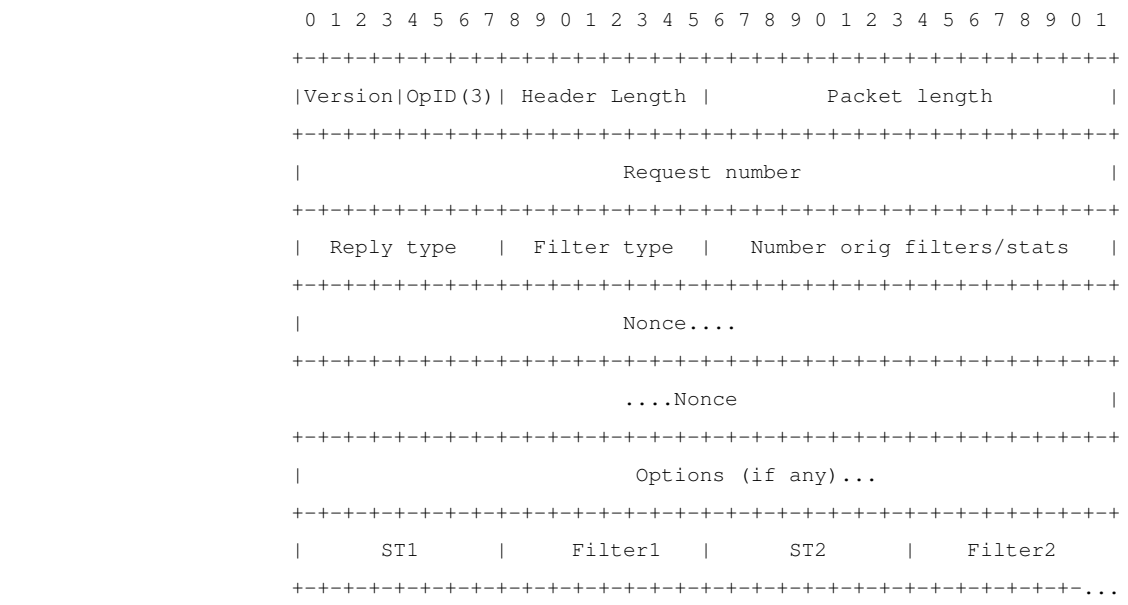

```
Nonce: 64 bits
   Used to ensure that the issuer of a request has not spoofed its IP
   address.
```

A.2.4 Nonce Reply (op id=3)

```
    The nonce reply packet can have two formats depending on the value
    of the Reply type field.

    Filter install/remove request header (see op id 0 for a description
    of the fields following the options):

     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```



Statistics request header (see op id 6 for a description of the fields following the options):



RT (reply type): 8 bits
Indicates which type of request prompted the nonce.

(0) Filter installation/removal
(1) Statistics request

FT (filter type): 8 bits
Indicates the type of filters contained in the packet (see above for details).

FET (filter expiration type): 4 bits
Indicates the type of timeout to apply to all filters in the packet. (see above for values).

AT (action type): 4 bits
Indicates the action to take when a packet matches the filter (see above for values).

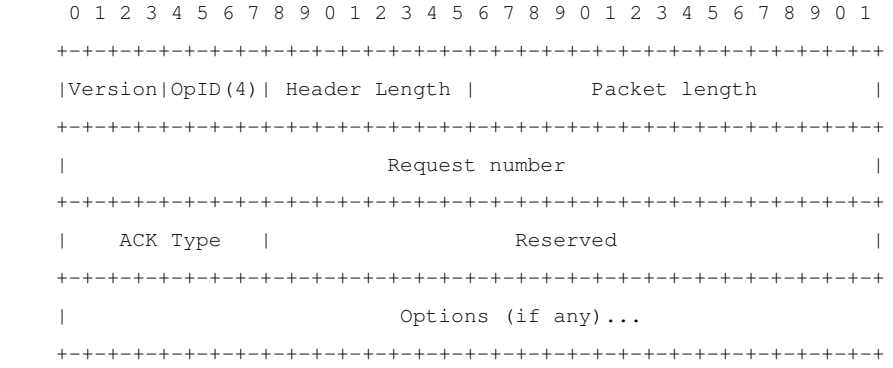
FEV (Filter expiration value): 16 bits
The value associated with the FET code specified (see above).

ATV (action type value): 8 bits
Contains the value associated with the AT code specified (see above).

NOF (number of original filters/stats): 16 bits
The sender of the original request is allowed to send other filters or statistics requests in addition to the original ones using this packet type. However, it must be assumed that the issuer of the nonce used the original filters/stats to compute the nonce, and so the nonce's recipient must indicate which filters/stats form part of the original request and which ones are being added.

Nonce: 64 bits
Used to ensure that the issuer of a request has spoofed its IP address.

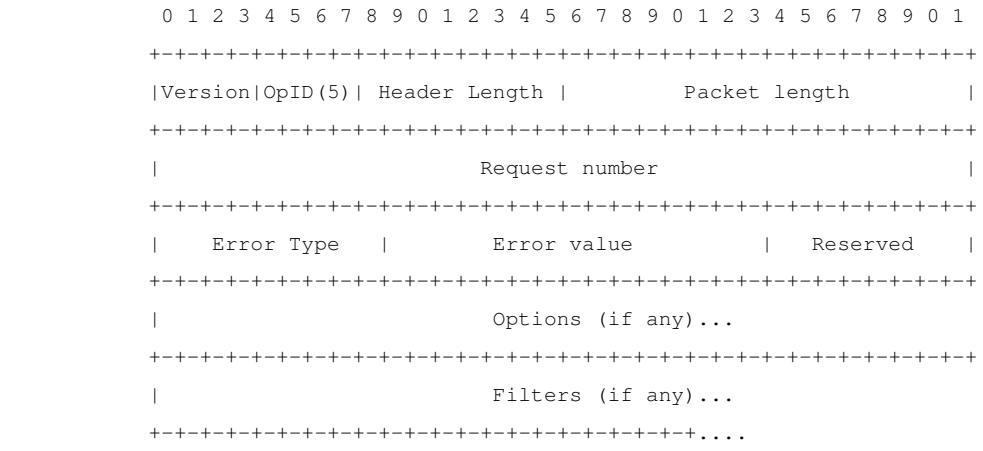
A.2.5 Ack (op id=4)



ACK Type: 8 bits
Indicates the type of action that the packet is acknowledging. The following codes are supported:

(0) Installed filters.
(1) Removed filters.

A.2.6 Error Reply (op id=5)



Error Type: 8 bits

Indicates the type of error that the packet is reporting. The following codes are supported:

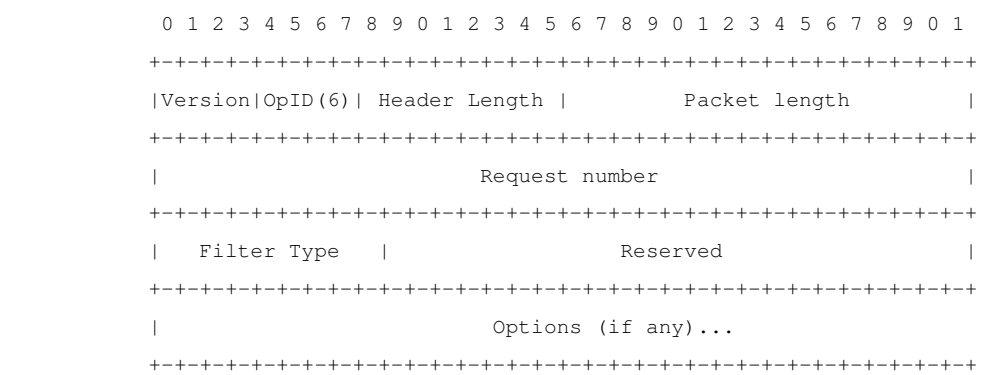
(0) Not responsible for given sources, cannot install/remove filters. Filters in packet represent those that could not be installed. EV field has filter type code.
(1) Too many filters, cannot install. Filters in packet represent those that could not be installed. EV field has filter type code.
(2) Please resend request in EV seconds.
(3) Protocol version not supported.
(4) Incorrect nonce received.

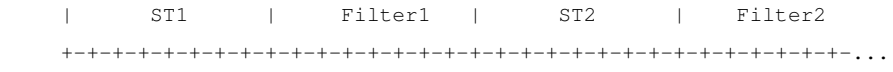
EV (Error Value): 16 bits

The value associated with the Error Type code specified (see above).

A.2.7 Statistics Request (op id=6)

This command is used to request statistics about particular filters. One packet may request different statistics from various filters, although all filters in the packet must be of the same type. Note that the fields labeled FilterN are variable, even though they're shown below to be 8 bits long.





FT (filter type): 8 bits
Indicates the type of filters contained in the packet (see above for details).

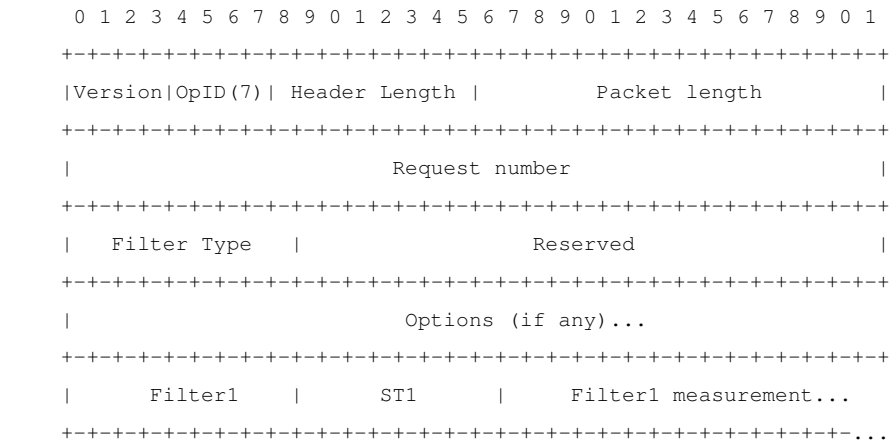
ST (Statistic type): 8 bits
Indicates the type of measurement that is required for each of the filters in the packet. Supported types are:

(0) Packet count since filter installation.
(1) Packet count since last reading.
(2) Current rate in packets per second. The reply will contain two fields, one for the number of packets, and the other for the measurement time window, in seconds.
(3) All of the above.

Filters: variable size
The filters to get statistics for.

A.2.8 Statistics Reply (op id=7)

This packet consists of a list of filters, each followed by the value of the measurement done for that particular filter.



FT (filter type): 8 bits
Indicates the type of filters contained in the packet (see above for details).

Filter: variable
See above for filter types.

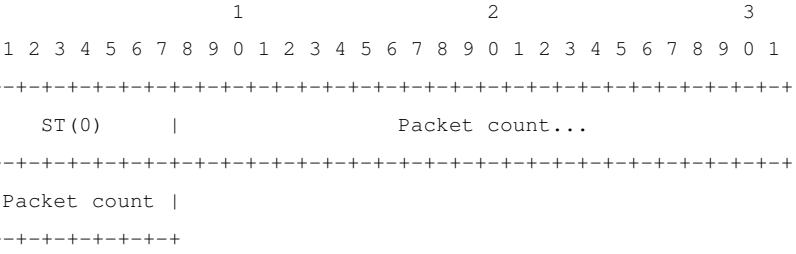
ST (Statistic type): 8 bits

Indicates the type of measurement that is required for each of the filters in the packet (see above for details). Please note that a measurement field value of all 1s will indicate that the host containing the filters does not support the statistic type requested.

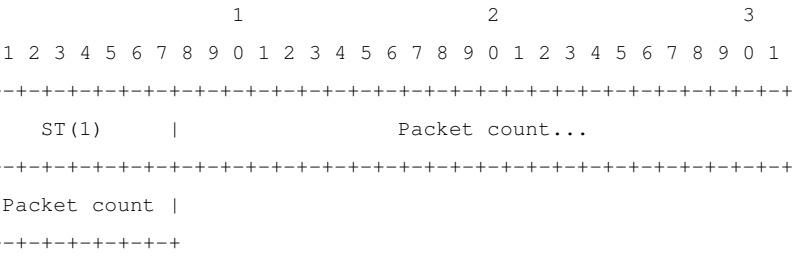
Filter measurement: variable

The length and format of the measurement field depends highly on the value of the ST type:

(0) Packet count since filter installation.

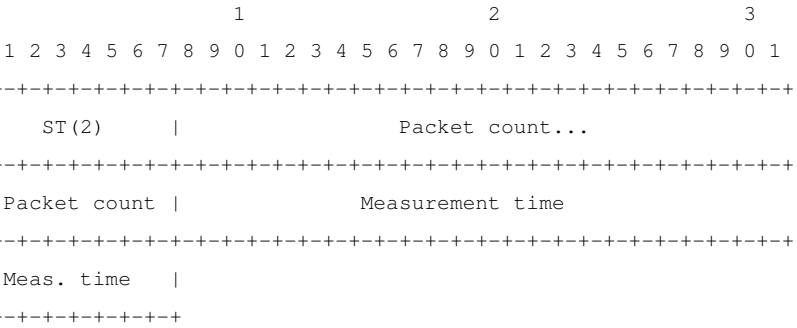


(1) Packet count since last reading.



The count used for the number of packets is 4 bytes long. The sender of the statistics request packet should be aware that this value might wrap.

(2) Current rate in packets per second.



The count used for the number of packets is 4 bytes long. The sender of the statistics request packet should be aware that this value might wrap. The measurement time is 4 bytes long and is assumed to be in

```
   seconds. The rate is simply the packet count divided by the measurement
   time.

(3) All of the above.

    0                   1                   2                   3
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   |     ST(3)     |    Packet count since filter installation...
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   ...Packet count |    Packet count since last reading...
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   ...Packet count |    Rate packet count...
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   ...Packet count |              Measurement time
   +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-++-+
   ...Meas. time   |
   +-+-+-+-+-+-+-+-+
```

Bibliography

[1] D. Adkins, K. Lakshminarayanan, A. Perrig, and I. Stoica. Brief announcement: towards a secure indirection infrastructure. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 383–383. ACM Press, 2004.

[2] D. Adkins, K. Lakshminarayanan, A. Perrig, and I. Stoica. Taming IP packet flooding attacks. *SIGCOMM Comput. Commun. Rev.*, 34(1):45–50, 2004.

[3] D. Adkins, S. Shenker, I. Stoica, S. Surana, and S. Zhuang. Internet Indirection Infrastructure. In *Proceedings of the ACM SIGCOMM 2002 Conference*, August 2002.

[4] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet Denial-of-Service with Capabilities. In *Proc. ACM SIGCOMM 2nd Workshop on Hot Topics in Networks*, November 2003.

[5] Arbor Networks. DDoS Attack Protection, Service Provider Network Visibility, Peakflow SP. http://www.arbornetworks.com/peakflowsp, 2008.

[6] K. Argyraki and D. Cheriton. Loose source routing as a mechanism for traffic policies. In *FDNA '04: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 57–64. ACM Press, 2004.

[7] K. Argyraki and D. Cheriton. Active Internet Traffic Filtering: Real-Time Response to Denial-of-Service Attacks. In *Usenix Annual Techical Conference*. Usenix, April 2005.

[8] H. Balakrishnan, F. Dabek, F. Kaashoek, D. Karger, D. Liben-Nowell, R. Morris, and I. Stoica. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[9] BBC News. Estonia fines man for 'cyber war'. http://news.bbc.co.uk/nolpda/ukfs_news/hi/newsid_7208000/7208511.stm, January 2008.

[10] S. Bellovin. The Security Flag in the IPv4 Header. http://www.ietf.org/rfc/rfc3514.txt, April 2003.

[11] S. Bellovin, M. Leech, and T. Taylor. The ICMP traceback message. http://tools.ietf.org/html/draft-ietf-itrace-04, February 2003.

[12] CAIDA: Cooperative Association for Internet Data Analysis. Packet Sizes and Sequencing. http://learn.caida.org, 1998.

[13] CAIDA: Cooperative Association for Internet Data Analysis. Packet Length Distributions. http://www.caida.org, 2000.

[14] Cisco. Cisco Guard DDoS Mitigation Appliances. http://www.cisco.com/en/US/products/ps5888/index.html, 2007.

[15] Cisco Systems. What is administrative distance? http://www.cisco.com/warp/public/105/admin_distance.html.

[16] Brad Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, May 2003.

[17] Intel Corporation. PCIe* GbE Controllers Open Source Software Developer's Manual, 2008.

[18] E. Felten, A. Halderman, A. Juels, and B. Waters. New client puzzle outsourcing techniques for DoS resistance. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 246–256. ACM Press, 2004.

[19] W. Feng, W. Feng, and A. Luu. The design and implementation of network puzzles. In *Proceedings of the 24th INFOCOM Conference*, March 2005.

[20] Force10 Networks. Force10 Networks: E-Series. http://www.force10networks.com/products/eseries.asp.

[21] Paul Francis. Firebreak: An IP Perimeter Defense Architecture. http://www.cs.cornell.edu/People/francis/hotnets-firebreak-v7.pdf, 2004.

[22] R. Govindan, A. Hussain, R. Lindell, J. Mehringer, and C. Papadopoulos. COSSACK: Coordinated suppression of simultaneous attacks. In *Proceedings of IEEE DISCEX Conference*, April 2003.

[23] A. Greenhalgh, N. Egi, M. Hoerdt, F. Huici, L. Mathy, and M. Handley. Fairness Issues in Software Virtual Routers. In *PRESTO '08: Proceedings of the ACM SIGCOMM workshop on Programmable Routers for Extensible Services of TOmorrow*. ACM Press, 2008.

[24] Mark Handley and Adam Greenhalgh. Steps towards a DoS-resistant Internet architecture. In *Workshop on Future Directions in Network Architecture (FDNA 2004)*. ACM SIGCOMM, September 2004.

[25] Mark Handley and Adam Greenhalgh. The case for pushing DNS. In *Proc. Hotnets IV*, November 2005.

[26] Y. He. How Asymmetric is Internet Routing? A Systematic Approach. In *Proceedings of the ACM SIGCOMM 2005 Conference (poster)*, August 2005.

[27] C. Jin, H. Wang, and K. Shin. Hop-count filtering: an effective defense against spoofed DDoS traffic. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 30–41. ACM Press, 2003.

[28] Juniper Networks. Tunnel services PIC datasheet. http://www.juniper.net/products/modules/tunnel_pic.html.

[29] B. Kantor and P. Lapsley. Rfc977, network news transfer protocol, a proposed standard for the stream-based transmission of news. ftp://ftp.ietf.org/rfc/rfc977.txt, February 1986.

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

[31] C. Kreibich, A. Warfield, J. Crowcroft, S. Hand, and I. Pratt. Using Packet Symmetry to Curtail Malicious Traffic. In *Proceedings of the 2005 HotNets Workshop*, 2005.

[32] W. Lee and J. Xu. Sustaining availability of web services under distributed denial of service attacks. *IEEE Transactions on Computers*, 52(3):195–208, 2003.

[33] J. Li, J. Mirkovic, M Wang, P Reiher, and L. Zhang. SAVE: Source address validity enforcement protocol. In *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001.

[34] Xin Liu, Ang Li, Xiaowei Yang, and David Wetherall. Passport: Secure and Adoptable Source Authentication. In *5th USENIX Symposium on Network Systems Design and Implementation*, April 2008.

[35] Ratul Mahajan, Steven M. Bellovin, Sally Floyd, John Ioannidis, Vern Paxson, and Scott Shenker. Controlling high bandwidth aggregates in the network. *SIGCOMM Comput. Commun. Rev.*, 32(3):62–73, 2002.

[36] Ajay Mahimkar, Jasraj Dange, Vitaly Shmatikov, Harrick M. Vin, and Yin Zhang. dFence: Transparent Network-based Denial of Service Mitigation. In *NSDI*. USENIX, 2007.

[37] Mazu Networks. Product Info - Mazu Profiler. http://www.mazunetworks.com/products/mazu-nba.php, 2008.

[38] MessageLabs. MessageLabs Intelligence: 2007 Annual Security Report. http://www.messagelabs.com/mlireport/MLI_Report_November_2007.pdf, 2007.

[39] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. In *Proceedings of the IEEE ICNP Conference*, November 2002.

[40] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. *SIGOPS Oper. Syst. Rev.*, 33(5):217–231, 1999.

[41] Network World. Extortion via DDoS on the rise. http://www.networkworld.com/, May 2005.

[42] K Park and H Lee. On the effectiveness of Route-Based packet filtering for distributed DoS attack prevention in Power-Law internets. In *Proceedings of ACM SIGCOMM*, pages 15–26, 2001.

[43] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks, 31(23-24)*, pages 2435–2463, December 1999.

[44] PCWorld. Hackers Hit Scientology With Online Attack. http://www.pcworld.com/article/id,141839-c,hackers/article.html, January 2008.

[45] PR-inside. IntruGuard IG2000 successfully defends against regular and increasingly heavy attacks. http://www.pr-inside.com/intruguard-ig2000-successfully-defends-against-r425993.htm, February 2008.

[46] Prolexic Technologies. Prolexic Technologies: Home. http://www.prolexic.com/, 2008.

[47] M. Reiter and X. Wang. Defending against denial-of-service attacks with puzzle auctions. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 78. IEEE Computer Society, 2003.

[48] Y. Rekther and T. Li. A border gateway protocol (BGP-4). http://www.ietf.org/rfc/rfc1771.txt, March 1995.

[49] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. *SIGCOMM Comput. Commun. Rev.*, 30(4):295–306, 2000.

[50] Security Focus. Blue Security folds under spammer's wrath. http://www.securityfocus.com/news/11392, May 2006.

[51] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM.

[52] Rishi Sinha, Christos Papadopoulos, and John Heidemann. Internet packet size distributions: Some observations. Technical Report ISI-TR-2007-643, USC/Information Sciences Institute, May 2007. Orignally released October 2005 as web page http://netweb.usc.edu/~rsinha/pkt-sizes/.

[53] Slashdot. 40-Gbps DDoS Attacks Worry Even Tier-1 ISPs. http://it.slashdot.org/it/08/11/11/192230.shtml, November 2008.

[54] A. Snoeren, C. Partridge, A. Sanchez, Jones C., F. Tchakountio, S. Kent, and T. Strayer. Hash-based IP traceback. In *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001.

[55] Robert Stone. Centertrack: an ip overlay network for tracking dos floods. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 15–15, Berkeley, CA, USA, 2000. USENIX Association.

[56] Symantec Corporation. Internet Security Threat Report Volume XI. http://www.symantec.com/enterprise/threatreport/index.jsp, March 2007.

[57] UCL Network Research Group. HEN: Heterogeneous Experimental Network. http://hen.cs.ucl.ac.uk.

[58] Patrick Verkaik, Oliver Spatscheck, Jacobus Van der Merwe, and Alex C. Snoeren. Primed: community-of-interest-based ddos mitigation. In *LSAD '06: Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 147–154, New York, NY, USA, 2006. ACM.

[59] W3C. W3C's Excessive DTD Traffic. http://www.w3.org/blog/systeam/2008/02/08/w3c_-s_excessive_dtd_traffic, February 2008.

[60] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS Defense by Offense. In *ACM SIGCOMM 2006*, 2006.

[61] A. Yaar, A. Perrig, and D. Song. Pi: A path identification mechanism to defend against DDoS attacks. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 93. IEEE Computer Society, 2003.

[62] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless internet flow filter to mitigate DDoS flooding attacks. In *Proceedings of the IEEE Security and Privacy Symposium*, May 2004.

[63] X. Yang. NIRA: a new internet routing architecture. In *FDNA '03: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 301–312. ACM Press, 2003.

[64] D. Yau, J. Lui, and F. Liang. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. In *Proceedings of the IEEE IWQoS Conference*, May 2002.

[65] ZDNET. Bot herders may have controlled 1.5 million pcs. http://news.zdnet.com/2100-1009_22-5906896.html, October 2005.