dFence: Transparent Network-based Denial of Service Mitigation

Ajay Mahimkar, Jasraj Dange, Vitaly Shmatikov, Harrick Vin, Yin Zhang
Department of Computer Sciences, The University of Texas at Austin
{mahimkar, jasraj, shmat, vin, yzhang}@cs.utexas.edu

Abstract

Denial of service (DoS) attacks are a growing threat to the availability of Internet services. We present dFence, a novel network-based defense system for mitigating DoS attacks. The main thesis of dFence is *complete transparency* to the existing Internet infrastructure with no software modifications at either routers, or the end hosts. dFence dynamically introduces special-purpose middlebox devices into the data paths of the hosts under attack. By intercepting both directions of IP traffic (to and from attacked hosts) and applying stateful defense policies, dFence middleboxes effectively mitigate a broad range of spoofed and unspoofed attacks. We describe the architecture of the dFence middlebox, mechanisms for on-demand introduction and removal, and DoS mitigation policies, including defenses against DoS attacks on the middlebox itself. We evaluate our prototype implementation based on Intel IXP network processors.

1 Introduction

Denial of service (DoS) attacks pose a significant threat to the reliability and availability of Internet services. Consequently, they are emerging as the weapon of choice for hackers, cyber-extortionists [24], and even terrorists [26]. The widespread availability of attack tools [3] makes it relatively easy even for "script kiddies" to mount significant denial of service attacks.

Our goal is to design and build a transparent network-based defense system capable of mitigating a broad range of large-scale, distributed denial of service attacks directly inside the network, without requiring software modification at either routers, or end hosts. Such a system can be deployed by Internet service providers (ISPs) in *today's* Internet, providing on-demand protection to customers, including those who operate legacy servers, only when they experience an actual attack. It can also serve as a general platform on which new security services and defense mechanisms can be deployed at a low cost, with a single installation protecting a large number of customers.

The problem of detection and mitigation of denial of service attacks has received considerable attention. Despite a large body of research literature and availability of commercial products, effective protection against denial of service has remained elusive. There are several plausible reasons for this. Most of the proposed solutions require software modification at either routers, or end hosts, or both. This means that the defense is not transparent to the Internet service providers and/or their customers.

Solutions that are not directly compatible with the existing TCP/IP implementations and routing software are likely to face unsurmountable deployment obstacles. Even SYN cookies, which are backward-compatible and part of standard Linux and FreeBSD distributions, are not used by the vast majority of users because they are turned off by default. At the same time, waiting until the Internet is re-engineered to provide better resistance against denial of service is not a feasible option for users who need immediate protection.

Arguably, the main challenge in DoS research today is not only coming up with new defense methods, but also finding an effective way to deploy both existing and new defenses with no changes to the installed software base, and without any performance cost when denial of service activity is not happening.

This problem is not unique to denial of service. Many network attacks are relatively rare events: a given end host may experience a denial of service attack once every few months, or be exposed to a new worm once a year. Therefore, there is little incentive for the end host operator to deploy an expensive protection system or to modify the existing software, especially if the change affects normal network performance. Of course, *not* deploying a defense can have catastrophic consequences when the attack does happen. We solve this conundrum by providing technological support for a "group insurance service" that an ISP can offer its customers: an on-demand defense that turns on only when the customer is actually experiencing an attack, and otherwise has no impact on the network operation.

Our contributions. We present *dFence*, a novel DoS mitigation system based on a small number of special-purpose "middlebox" devices located in the middle of the network. The main features of dFence are:

- **Transparency:** dFence is fully transparent to the end hosts, requires no modifications of client or server software, and enables protection of legacy systems. Protected hosts need not even be aware that the system is in operation.

- **Compatibility with routing infrastructure:** dFence employs standard intra-domain routing and tunneling mechanisms for traffic interception. It requires no changes to the existing router software, and is thus incrementally deployable by Internet service providers.
- **On-demand invocation:** dFence middleboxes are dynamically introduced into the data path of network connections whose destinations are experiencing denial of service, and removed when the attack subsides. The small performance cost of filtering is paid *only* by the attacked hosts, and only for the duration of the attack.
- **Scalability:** The dynamic nature of dFence allows ISPs to multiplex the same defense infrastructure to protect a large number of customers (who are not all under attack *at the same time*), and thus more efficiently utilize their network and computing resources.
- **Minimal impact on legitimate connections:** Each dFence middlebox manages the state of active, legitimate connections to the customers who are simultaneously under attack. Malicious flows do not occupy any memory at the middlebox. Therefore, legitimate flows can be processed with very low latency cost. The cost for the flows to the destinations *not* experiencing an attack is zero.
- **Versatility:** Because dFence middleboxes dynamically intercept *both* directions of TCP connections to DoS-affected hosts, they can apply stateful mitigation policies to defend against the entire spectrum of DoS attacks.
- **Economic incentive:** We envision dFence middleboxes being deployed within a single ISP. The ISP can then charge a premium to customers who subscribe for a dFence-based "insurance service." dFence middleboxes are turned on only when one or more paying customers are experiencing an attack. As more customers subscribe to the service, the ISP can incrementally scale up the deployment.

System architecture. Figure 1 depicts the overall system architecture. The two guiding design principles behind dFence are *dynamic introduction* and *stateful mitigation*. We implement dynamic introduction by using intra-domain routing and tunneling mechanisms to transparently insert dFence middleboxes into the data path of traffic destined to hosts experiencing a DoS attack. This is done *only* when DoS activity is detected in the network. Due to dynamic introduction, our solution has zero impact on normal network operations, and can be deployed incrementally.

The middleboxes intercept both directions of network traffic (to and from attacked hosts), which enables many mitigation techniques that were previously considered unsuitable for network-based defenses. The main technical novelty is the network-based implementation of defenses that previously required modifications to server or client software. For example, "outsourcing" SYN cookie generation to dFence middleboxes enables us to protect legacy end hosts whose TCP/IP implementations do not support SYN cookies. Other mitigation techniques include defenses against spoofed and unspoofed data floods, against clients opening and abandoning a large number of connections, and against distributed botnet attacks.

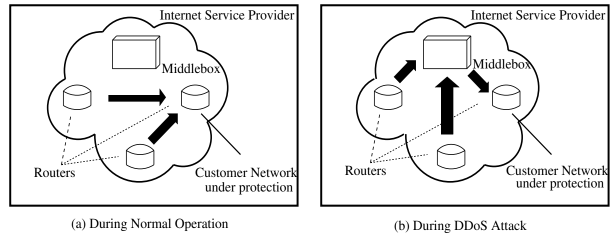


Figure 1: dFence Architecture: (a) During normal operation, the ingress routers forward the traffic towards the corresponding egress routers. (b) Under a large-scale DoS attack, traffic is re-directed via the middlebox. The middlebox applies mitigation policies and filters out illegitimate traffic.

Key challenges. The combination of transparent on-demand defense, two-way traffic interception, and stateful mitigation presents several interesting challenges: (i) how to deal with middlebox transitions, *i.e.*, how to introduce and and remove middleboxes on selected data paths; (ii) how to dynamically bootstrap, manage, and remove connection state at the middleboxes; (iii) how to handle network behavior such as route changes and failures of network elements; and (iv) how to handle overload conditions and DoS attacks on the middleboxes themselves.

We present practical solutions to all of these challenges. Our main contribution is a careful integration of several network mechanisms leading to a *completely transparent, scalable and effective DoS mitigation system*. We evaluate our design using a prototype implementation based on Intel's IXP2400 network processors, and demonstrate that mitigation of a broad range of DoS attacks, including spoofed SYN floods and unspoofed data floods, can be achieved with minimal performance degradation. We also note that the focus of this paper is on mitigation, rather than detection of denial of service activity.

Organization. The rest of the paper is organized as follows. In Section 2, we describe the mechanisms for dynamically introducing dFence middleboxes in the data path of attack traffic. Section 3 describes the architecture of the dFence middleboxes and the mitigation policies that defend against a broad range of spoofed and unspoofed attacks. Our prototype implementation and its performance evaluation are presented in Sections 4 and 5, respectively. Section 6 outlines the related work. Section 7 summarizes our contributions.

2 Transparent Middlebox Invocation

A key design principle for dFence middleboxes is complete transparency to the end hosts. This is achieved through *dynamic invocation* of middleboxes by standard intra-domain routing mechanisms and tunneling. A few dFence middleboxes are introduced into the network to provide focused protection to the subset of the end hosts that are currently experiencing a DoS attack. Protected hosts do not need to modify their software, nor access the network through special overlay points, nor set up IP tunnels, nor even be aware that dFence middleboxes have been deployed inside their ISP.

Dynamic middlebox invocation is critical for deployability because it ensures that during peace time (*i.e.*, when there is no ongoing DDoS activity) customer traffic does not have to pay the penalty of triangular routing through the middleboxes. Dynamic middlebox invocation is also important for the defense system itself because it focuses all defense resources only on the connections whose destinations are under attack, leaving other customers unaffected. The defense system can thus benefit from statistical multiplexing and potentially protect many more customer networks with the same available resources.

Combining dynamic middlebox invocation with stateful attack mitigation raises several technical challenges.

- *Bidirectional traffic interception.* Many of our mitigation policies require the defense system to capture both directions of customer traffic. For example, to protect against spoofed data floods from remote hosts to a customer network under protection, we maintain a Connection table to summarize on-going TCP connections. Bidirectional traffic interception is difficult in general because Internet routing is destination-based by default; intercepting traffic from the customer network, however, requires the ability to perform *source*-based routing.

- *Flow pinning.* In addition to intercepting both directions of protected traffic, stateful mitigation also requires that both directions pass through the *same* middlebox (where the state of the connection is maintained), even after routing changes have caused the intercepted traffic to use different ingress points. This requires a mechanism for pinning each flow, defined by the TCP/IP packet header, to a particular middlebox. Flow pinning also provides security against an attacker who attempts to disrupt external routing while launching an attack.

  We use a simple hash-based flow pinning method. Each flow is associated with a *home* middlebox, whose identity is determined by a hash $h(f)$ of the flow identifier f. The flow identifier consists of source and destination IP addresses and port numbers in the TCP/IP packet header. If the flow is intercepted by a *foreign* middlebox, it is simply forwarded to the home middlebox, achieving reasonable load balancing of flows across the middleboxes. In the future, we plan to investigate more sophisticated flow pinning mechanisms.

- *Dynamic state management.* Dynamic middlebox invocation also poses interesting challenges for state management. The first question is how to handle existing connections when the middlebox is inserted into the data path. For instance, our policy for defending against spoofed data flooding can drop a data packet if it is not in the Bloom filter summary of ongoing connections. But an existing legitimate connection may not be present in the Bloom filter if it had been established before the middlebox was introduced. We would like to minimize the number of packets dropped for such connections.

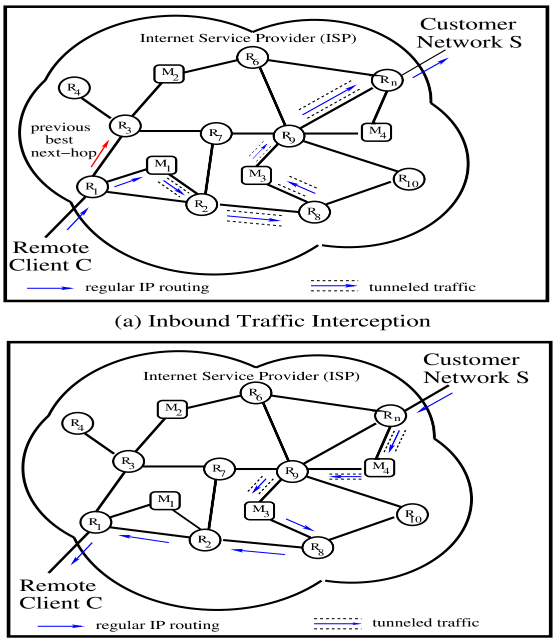
  Besides filtering, some of our advanced mitigation policies perform operations that change the content of the traffic. For example, the "outsourced" SYN cookie policy requires the splicing of TCP connections and sequence number translation to be performed at the middlebox. What happens to the spliced connections when the middlebox is removed from the data path?

- *Middlebox failure recovery.* A dFence middlebox may fail due to software or hardware errors, or traffic overload. Protecting the overall defense system from middlebox failures is an important challenge.

Our solution is based on standard BGP/IGP routing, tunneling, and policy-based routing (PBR) [7], which is available in almost all existing routers. Therefore, our solution is easy to deploy in today's Internet. In addition, we implement a simple hash-based mechanism for pinning each individual connection to the same *home* middlebox. This ensures that both directions of the connection traverses the same middlebox (even in the presence of route changes that may result in different ingress points). Below we present our approach for bidirectional traffic interception, dynamic state management, middlebox failure recovery and load balancing.

2.1 Dynamic Traffic Interception

While there exist several academic and industrial solutions for traffic interception [1, 8, 11], none of them, to the best of our knowledge, can simultaneously (i) introduce middleboxes dynamically, (ii) intercept both inbound and outbound traffic, and (iii) ensure that both directions of a connection go through the same home middlebox. As a result, no single existing technique is sufficient for stateful attack mitigation. It is possible, however, to use existing techniques to substitute some of the individual components of our solution (*e.g.*, our mechanism for inbound traffic interception).



(a) Inbound Traffic Interception

(b) Outbound Traffic Interception

Figure 2: Traffic interception at middleboxes using BGP/IGP, tunneling and policy-based routing

2.1.1 Inbound traffic interception

In our prototype implementation, we use iBGP and tunneling to intercept inbound traffic (alternatives include mechanisms such as MPLS tunneling). To intercept all traffic inbound to some customer network S, the dFence middleboxes send iBGP updates advertising a route to S with the highest local preference to all BGP peers in the local AS. As a result, the middleboxes become the preferred egress points for all traffic destined to S. At each ingress router, IGP selects the closest middlebox and updates the forwarding tables appropriately.

To enable the packets to reach S after they have been filtered by the middlebox, dFence configures a tunnel from the middlebox to the real egress router associated with S. The tunnel can be set up using any available mechanism; in our prototype, we use IP-IP tunnels. The egress router specifies two ACLs: (a) *ACL-to-S* is defined on the router's internal interface (connecting it to the rest of the ISP) and intended for traffic going towards S; (b) *ACL-from-S* is defined on the external interface connecting it to the customer network and intended for traffic arriving from S.

The journey of an incoming packet typically consists of the following three steps, as illustrated in Figure 2(a).

1. *Go to one of the middleboxes:* IGP selects the middlebox M_1, which is the closest middlebox to the ingress point. If M_1 is the home middlebox for this flow, the next step is skipped; otherwise M_1 needs to forward the packet to its home middlebox.

2. *Flow pinning:* The packet is tunneled from the foreign middlebox M_1 to the home middlebox M_3. The identity of the home middlebox is determined by the hash value of the flow identifier. The home middlebox M_3 then applies mitigation policies described in section 3.2 to process the packet.

3. *Go to the real egress router:* After the middleboxes advertised routes to S, the intermediate routers' forwarding tables point towards middleboxes for all packets whose destination is S. To avoid routing loops, we tunnel the packet from the home middlebox M_3 to the true egress router R_n. When R_n receives the tunneled packet, it decapsulates the packet and, because it matches *ACL-to-S*, forwards it to the customer network S.

Observe that the traffic arriving on the *external* interfaces of R_n (other than the interface connecting it to S) will be first re-routed to the middlebox for filtering, because the middleboxes' iBGP route advertisements change the forwarding table at R_n, too.

2.1.2 Outbound traffic interception

We use policy-based routing (PBR) [7] to intercept outbound traffic originating from the customer network S to a remote host. PBR allows flexible specification of routing policies for certain types of packets, including ACLs (access control lists) that identify classes of traffic based on common packet header fields. In our context, we use PBR to forward all traffic that matches *ACL-from-S* to a dFence middlebox through a preconfigured tunnel.

The journey of an outbound packet consists of the following three steps, as illustrated in Figure 2(b).

1. *Go to one of the middleboxes:* When the egress router R_n receives a packet from S, the flow definition matches *ACL-from-S* and so the packet is forwarded to middlebox M_4 through its preconfigured tunnel interface.

2. *Flow pinning:* M_4 forwards packets to the home middlebox M_3, determined by the flow pinning hash function. The hash function is selected in such a way that exchanging source and destination fields does not affect the hash value, for example:

$$h_1(src\_addr, src\_port) \oplus h_2(dst\_addr, dst\_port)$$

   where h_1 and h_2 are two independent hash functions. This ensures that both directions of the same connection have the same home middlebox (M_3 in our example)

3. *Go towards the egress router*: Regular IP routing is used to send the packet to its egress router R_1.

2.2 Dynamic State Management

The main challenge in dynamic middlebox invocation is the need to gracefully handle existing connections upon the introduction or removal of a middlebox. Our basic solution is to add *grace periods* after the introduction or before the removal of the middlebox. During the grace

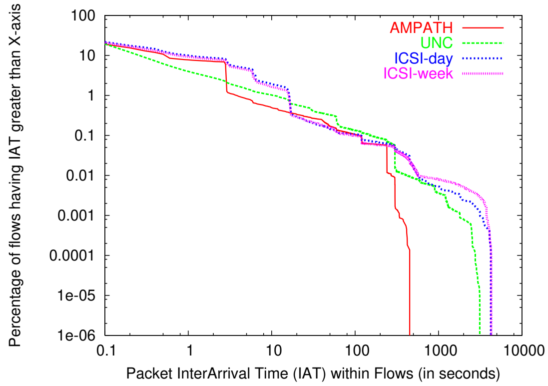


Figure 3: Packet inter-arrival time (IAT) within flows

period, the middlebox continues to serve all existing connections while it is preparing to establish or time out its state.

After the middlebox has been introduced into the data path, it spends T_b seconds (*state bootstrap interval*) bootstrapping its state. After the decision has been made by the network operator to remove the middlebox, the middlebox stays in the data path for another T_r seconds (*state removal interval*) before being completely removed.

1. *State bootstrapping:* During interval T_b, the middlebox establishes state for the existing connections between clients and the server and/or customer network which is being protected. An existing connection is considered legitimate if the middlebox sees both inbound and outbound traffic on it during the bootstrap period. The list of connections is maintained using the Connection table data structure, described in detail in Section 3.1.

2. *State removal:* After the removal decision has been made, the middlebox can be removed right away if no currently active mitigation policy involves modifications to individual packet content. Some mitigation policies, however, such as "outsourced" SYN cookie generation cause the middlebox to actively modify packet headers (*e.g.*, by sequence number translation). For these policies, the middlebox cannot be removed right away because the existing connections can become invalid without the translations performed by the middlebox. Therefore, the middlebox needs to remain in the data path during interval T_r and continue to serve the ongoing connections. No policies are applied on any new connections and they are directly forwarded to their destinations.

The value of the state bootstrap interval T_b is important. If T_b is too long, then the attacker can cause severe damage while the middlebox is being bootstrapped. If T_b is too short, then many existing connections may either get terminated, or suffer poor performance. Trace analysis based on several large datasets shows that the vast majority (99%) of all connections have packet interval times that are quite small, on the order of a few seconds (see Figure 3). Hence, T_b can be set to 5 or 10 seconds. This means that, within a short bootstrap interval, the state for the vast majority of existing connections can be established, and only a handful of legitimate users (those who were idle during T_b) will have to re-establish connections.

It might appear that the bootstrap time T_b provides an opportunity for an attacker to overload the middlebox itself. This is not the case because connection state is maintained at the middlebox only for unspoofed connections which comply with traffic control measures.

The decision to introduce a middlebox can be made by the customer networks under protection (*e.g.*, when they observe too much inbound traffic), or through some network-based DoS detection system. Since our primary focus in this paper is on attack mitigation, we do not discuss attack detection here.

The value of the removal interval T_r can be prespecified (it must be sufficiently long to allow for normal termination of all ongoing connections), or it can be adaptive based on the number of connections going through the middlebox. Compared with T_b, the choice of T_r is less critical because it primarily affects the amount of additional work that the middlebox performs and has little impact on the safety of customer networks.

The decision to remove a middlebox can only be made by the middlebox itself (or by all middleboxes collectively). Unlike middlebox introduction, middlebox removal cannot be decided by the customer networks—if the defense system is effective, then the customers should not be able to tell whether the DoS attack ceased, or the attack is still in progress. Therefore, the middleboxes need to continuously profile the (unfiltered) traffic and decide whether the ongoing attack has subsided.

2.3 Failure Recovery and Load Balancing

A middlebox can fail for a variety of reasons, such as power outage, hardware malfunction, software errors, network outages, and so on. It can also fall victim to a DoS attack itself, even though our middlebox architecture is explicitly hardened against this possibility. When a middlebox fails, it is crucial that a different middlebox (or the same middlebox after rebooting) take over the management of all connections whose flow identifiers have been pinned to the failed middlebox. To make this transition as smooth as possible, the middlebox must offload its state to a different middlebox as soon as overloading is detected. Therefore, *graceful flow migration* is the key component of both failure recovery and load balancing. We outline how it can be achieved in the dFence mitigation system.

Recall that each flow identifier f is pinned to its home middlebox by a hash function $h(f)$. To avoid changing the hash function (which may be implemented in hardware and difficult to modify), we introduce one level of indirection. All middleboxes in our system will agree on a global *home middlebox table* $HM[0..n-1]$ (*e.g.*,

$n = 1024$). Each middlebox is responsible for a subset of entries in this table, or, more precisely, for all flows whose identifiers hash to this subset. The global *HM* table can be maintained either centrally, or through a distributed agreement protocol among all the middleboxes. The hash function h can be chosen so that it maps flow identifier f to $\{0, 1, \cdots, n-1\}$. The home middlebox for flow f is simply *HM*$[h(f)]$. This enables graceful flow migration for failure recovery and load balancing.

Failure recovery. All middleboxes are pre-arranged to form a logical ring R. For each middlebox M_i, its clockwise next-hop neighbor in R (denoted by $R.\text{next}(M_i)$) is the designated backup for M_i and will take over the flows managed by M_i should M_i fail. Suppose for some entry e the original home middlebox $M_1 = HM[e]$ failed. $M_2 = R.\text{next}(M_1)$ then becomes the new $HM[e]$ and starts the bootstrap interval T_b, during which it bootstraps (as described in section 2.2) the state of all ongoing connections whose flows are hashed to e. The same procedure can be repeated to handle multiple failures. For example, if both M_1 and M_2 failed, then $M_3 = R.\text{next}(M_2)$ becomes the new $HM[e]$. Note that unreachable middleboxes due to network partition can be handled in the same way as failures. The only additional processing required for coping with network partition is to resolve the inconsistency in the *HM* tables maintained by different partitions after the network becomes connected again.

Load balancing. Load balancing can be handled in a similar manner. Suppose M_1 is overloaded and wants to offload all flows that are hashed to entry e to a less loaded middlebox M_2. All it needs to do is to update the global home middlebox table so that $HM[e] = M_2$. M_2 then spends the T_b period to bootstrap its state for flows that are mapped to e. Note that during the state bootstrap interval, instead of blindly letting through every flow that M_2 has no state for, M_2 has the option of forwarding such flows to M_1. This can make flow migration more graceful, especially when M_1 has been applying traffic-modifying mitigation policies such as SYN cookies and sequence number translation.

3 Middlebox Design

3.1 Overview

dFence is based on diverting traffic to special-purpose *middleboxes* as soon as denial of service activity is detected. Each middlebox is responsible for protecting TCP connections to some or all of the attacked destinations. To effectively distinguish between benign and malicious traffic, the middlebox maintains partial TCP state for both directions of the intercepted connections, but does not buffer any packets. Because mitigation is performed entirely within the middlebox and traffic redirection is achieved using standard intra-domain routing and tunneling mechanisms, dFence does not require any software modification at either routers, or the end hosts. We focus on TCP-based attacks, but UDP, ICMP or DNS attacks could be handled in a similar fashion.

TCP connection management. TCP connections managed by a dFence middlebox include pre-existing connections that had been established before the middlebox was introduced into the data path (the middlebox acquires these connections during the bootstrap period—see section 2.2), and those established after the middlebox became active. The latter connections are *spliced* to enable anti-spoofing defenses. Splicing performed by the middlebox is very simple and limited to translation of sequence numbers.

The main data structure maintained by the middlebox is the Connection hash table, which tracks the state of all established connections (both directions). Entries in the table are identified by the hash of *FlowId*, which consists of the source IP address, destination IP address, source port, and destination port. Each entry includes the following:

- *Flow definition*: source IP, destination IP, source port, destination port. [4 bytes per IP, 2 bytes per port; 12 bytes total]
- *Offset*: The difference between sequence numbers on the middlebox-source connection (generated as SYN cookies by the middlebox) and the destination-middlebox connection (chosen by the destination when a connection is established by the middlebox on behalf of a verified source). This offset is used to translate sequence numbers when the two connections are "spliced" at the middlebox. [4 bytes]
- *Timestamp*: Last time a packet was seen on this connection. Used to time out passive connections. [4 bytes]
- *Service bits*: (i) *pre-existing*: is this a pre-existing or spliced connection? (ii) *splice*: is sequence number translation required? (iii) *conformance*: has the source complied with traffic management measures (*e.g.*, responded properly to congestion control messages)?
- *InboundPacketRate*: Array of size $\frac{T}{t_i}$, containing the number of inbound packets seen for each interval of length t_i (T is the monitoring period). Used to mitigate unspoofed data flood attacks.

Preventing resource exhaustion and resolving collisions. To prevent the attacker from filling the Connection table with a large number of connection entries that have the same (unspoofed) source and destination IP addresses, but different port numbers, the middlebox maintains a separate Src-Dest table. This is a hash table indexed by the hash of the source IP - destination IP pair. For each pair, it keeps the count of currently open connections. Once the threshold is exceeded, no new connections are established. The value of the threshold is a system parameter, and can be changed adaptively depending on how full the Connection table is.

To resolve hash collisions between different flows in the Connection table, we use a Bloom filter-like technique and apply several hash functions until a vacant entry is found. If no vacancy can be found, the decision whether to drop the new flow or evict the old flow depends on the status of the latter and specific policy (see section 3.2).

The middlebox also maintains a secret key which is used as input to a hash function for generating unforgeable sequence numbers. This key is the same for all connections. It is re-generated at periodic intervals.

Handling connections originated from a protected network. In addition to keeping information about connections whose destination is the protected server, the middlebox also needs to keep information about the connections originating *from* the server in order to filter out packets with spoofed server address. This is done using a sliding-window counting Bloom filter [5]. (In our current implementation, we use filters with 3 hash functions.) During each time slice t_i, when a connection request from the server is observed, the middlebox adds connection parameters to the current Bloom filter B_i. If connection is terminated, it is removed from B_i.

3.2 Mitigation Policies

A large number of techniques for mitigating various types of DoS attacks have been proposed in the research literature. Virtually none have been deployed widely, due mainly to the lack of *transparency* and *scalability*: networking software must be modified at millions of end hosts, and performance penalty must be paid even when the hosts are *not* being attacked. Moreover, different attack types require different defenses, and supporting all of them (SYN cookies, capabilities, client puzzles, and so on) in a general-purpose TCP implementation is neither feasible, nor desirable.

Our main technical contribution in this part of the paper is to show how many anti-DoS defenses can be effectively and transparently implemented in the middle of the network at a minimal performance cost.

3.2.1 Mitigating spoofed attacks

The distinguishing characteristic of spoofing attacks is that the source addresses of attack packets are fake. For example, SYN flood is a classic denial of service attack, in which the attacker sends a large number of SYN requests to a TCP server. The server creates a half-open connection in response to each request. Once the server's queue fills up, all connection attempts are denied. In a spoofed data flood, the attacker simply floods last-mile bandwidth with spurious traffic. In Smurf-type and reflector attacks, the attacker sends packets with the victim's address in the source field to a large number of hosts, who then all respond to the victim, overwhelming him with traffic.

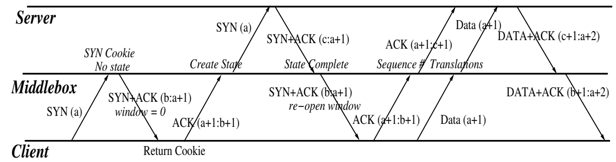


Figure 4: Outsourced SYN cookies with sequence number translation

Network-based SYN cookie generation. Our policy for mitigating spoofed SYN floods is shown in fig. 4. It is based on the well-known idea of *SYN cookies* [4, 20], except that, unlike the traditional approach, we do not require *any* modifications to the server TCP software.

After a dFence middlebox M has been dynamically introduced into all routes to some host S that is experiencing a denial of service attack, all traffic to S passes through M. On receipt of a SYN packet whose destination is S, the middlebox computes the SYN cookie as a cryptographic hash of connection parameters and the (frequently re-generated) local secret, adds the value of the cookie to the sequence number in the SYN packet, and uses it as the sequence number in its SYN-ACK response. No state is established at the middlebox at this stage.

Note that in the SYN-ACK response, *the middlebox M sets the receiver window size to zero*. Upon receiving the SYN-ACK with zero window size, C sends back an ACK packet and then enters the TCP Persist Mode. While in this state, C is not allowed to send any data packets with non-zero payload. So M effectively "chokes" C. M can "unchoke" C later by sending it any packet with a non-zero window size. If M receives data packets from C before the handshake with S is complete, it marks C as non-conforming and simply drops all further packets from C.

Note that, for a legitimate client that does not have correct implementation of persist mode, the middlebox will classify it as non-conforming and drop its packets.

It is important to prevent C from generating data packets before M completes its handshake with S (which may be far away from M). Otherwise, M has to buffer all these packets, which can be expensive. Dropping these data packets is not a good option because when the first few data packets at the beginning of a TCP connection are dropped, C can recover only through the TCP timeout mechanism. The default TCP timeout value is often set to 3 seconds, which can seriously degrade network performance as perceived by end users. We have confirmed this experimentally by turning off the choking mechanism, and observed a 3-second timeout each time. For server-side SYN cookies, choking is not needed because S will only receive packets from M *after* M has completed the handshake with a legitimate client, and thus all of S's packets can be safely forwarded to that client.

On receipt of an ACK packet from some client C, the middlebox M recomputes the cookie-based sequence number and verifies that it is correct. If so, the connection is not spoofed, and M creates new entries for it in the Connection and Src-Dest tables (see section 3.2). The entries are indexed by the hash of connection parameters. If a collision occurs, it is resolved as described in section 3.2.

At this point, M needs to establish a connection with the protected server S on behalf of the (verified) client C. This is done by performing the standard TCP handshake with S. M uses the same sequence number that C used in its SYN packet, by subtracting 1 from the sequence number in the ACK packet.

When M receives SYN-ACK from S, M forwards it to C and re-opens the window. There is a technical challenge here, however: the sequence numbers chosen by S for the $M - S$ connection are not the same as the cookie-based sequence numbers generated by M for the $C - M$ connection. As described in section 3.2, for every connection M maintains the offset between the two sequence numbers. On receiving SYN-ACK, C assumes that its previous ACK packet was lost and thus retransmits its ACK. C also exits the persistent mode as the SYN-ACK packet now has a non-zero receiver window size. M forwards ACK with proper sequence and acknowledgement numbers, thereby completing the handshake with S.

All subsequent data packets undergo sequence/ack number translation at M. When a packet arrives from S, M adds the offset to the sequence number. When a packet arrives from C, M subtracts the offset from the acknowledgement number. The *splice* bit is set in the Connection table to indicate that sequence number translation is required.

Spoofed data floods and reflector attacks. As described in section 3, the middlebox maintains information about client-originated connections (in the Connection table) as well as the connections originating from the server that is being protected (in the Bloom filter). Any data packet whose flow identification is not found in either of these two data structures is dropped. The same defense works against reflector attacks, because the middlebox filters out data packets from reflectors whose connection parameters do not belong to either the Bloom filter or Connection table.

3.2.2 Mitigating unspoofed attacks

Unspoofed data floods. The attacker can launch a data flood from a legitimate address by completing the TCP handshake and then flooding the bandwidth with data traffic. Our defense is based on enforcing compliance with congestion control measures.

When traffic rate on a connection exceeds some threshold value h (h is a tunable system parameter), the middlebox modifies ACK packets arriving from the server to reduce the receiver advertisement window, and starts measuring the rate of packets arriving from the client. (Recall from section 3 that the Connection table includes arrays for measuring packet rates on each unspoofed connection.) If at the end of the monitoring period the client's packet rate did *not* show a decreasing trend, the *conformance* bit is set to 0. All data packets on connections where the measurement period ended and the *conformance* bit is 0 are dropped. Note that this defense is feasible because a dFence middlebox controls *both* directions of TCP connections. The threshold h can be adaptively set based on the middlebox load.

Too many unspoofed connections. Many denial of service attacks involve the attacker opening a large number of connections from a legitimate IP address that belongs to a compromised, remotely controlled "zombie" machine. The zombie completes the TCP handshake, conforms to congestion control measures, and then overwhelms the server with a large number of requests. The Src-Dest table (see section 3) defends against multiple connections from the same address by limiting each source-destination pair to a reasonable number of connections.

NAPTHA attacks. In the NAPTHA attack, the attacker opens a legitimate connection, immediately closes it without sending FIN/RST, and opens another connection from a different zombie machine. This fills up the server's state, causing denial of service to legitimate connections.

To defend against NAPTHA attacks, the dFence middlebox maintains a timestamp for each connection, indicating the last time a packet was observed. If the idle time of a connection (the time since the last packet was observed) exceeds a threshold value (which is a tunable system parameter), the middlebox "times out" the connection by sending RST to the server. This is also done when the Connection table fills up, leading to a large number of collisions. The thresholds can also be determined empirically by analyzing attack traces.

Botnet attacks. dFence middleboxes can also support more sophisticated filtering policies. As an illustration, we sketch how they can be used to defend against botnet attacks. Our goal in this section is to demonstrate the expressiveness of dFence policies rather than describe a comprehensive solution to the botnet problem.

In botnet attacks, the attacker commands a large number of compromised computers to bombard the victim with HTTP or other requests. From the victim's viewpoint, this situation is similar to a flash crowd, since it is difficult to tell whether an individual connection is malicious or benign.

Our dFence-based botnet mitigation policy is based on source-prefix whitelisting. This policy is invoked only after the client has complied with all other measures, including congestion control. It gives preference to traffic from hosts in the white list of N most common /24 prefixes for a given server or network. This list can be created from traffic statistics, or else ISP customers can pay to be added to it.

The reason this approach is effective against botnets is that zombie machines tend to be sparsely distributed, *i.e.*, the attacker is likely to control only a handful of zombies within each /24 prefix. This observation is confirmed by our analysis of botnet traces collected by [9]. In both traces, fewer than 10 machines from any single /24 prefix are used during the attack. In trace I, 99% of prefixes have no more than 2 zombies, and in trace II, 99% of prefixes have no more than 7. In trace I, only 3 out of 22203 observed prefixes have more than 20 zombies, and in trace II, 48 out of 64667. (Note that the middlebox eliminates all spoofed connections using the anti-spoofing defenses described above, and that each bot is restricted to a modest amount of traffic by congestion control and compliance checking measures.)

This approach can be easily combined with an adaptive form of CAPTCHA-based Kill-bots [16]. The middlebox can adaptively redirect HTTP traffic from outside the preferred prefixes to a CAPTCHA server. This can be viewed as *rationing*: some fraction of the flows in the Connection table are allocated to the top N privileged flows, with the remaining (or un-privileged) flows competing for the rest of the table entries.

3.3 Policy Decision Logic

Because the dFence middlebox is in the data path of all connections to and from the servers that are being protected, it is critical to ensure that per-packet processing complexity is low and can scale to high link speeds. In particular, we want to avoid blindly applying different mitigation policies one after another regardless of the packet type.

On receipt of a packet, the middlebox first classifies it using TCP flag types. Depending on which flag is set (SYN, SYN+ACK, FIN, RST, *etc.*), it is sent to the respective processing function. For a SYN packet from client during the bootstrap or active phases, a SYN cookie is generated and SYN-ACK sent back to the client. For SYNs from server, the Bloom filter is updated. For SYN-ACKs from the server during the bootstrap or active phases, the Connection table is updated with the right offset value (difference between the seq/ack numbers on the middlebox-source and middlebox-server connections). During the removal phase, SYNs and SYN-ACKs are simply forwarded without updating the data structures at the middlebox.

For a data packet, its 4-tuple flow ID (IP addresses and port numbers) is looked up and checked against the Connection table and the Bloom filter to verify that it belongs to an established connection. If in the Bloom filter, the packets are forwarded. If in the Connection table, the *pre-existing* bit is checked and splicing performed, if needed. During the bootstrap phase, packets whose flow ID does *not* belong to both the Bloom filter and the Connection table are forwarded and middlebox state updated. During the active phase, they are assumed to be spoofed and dropped. During the removal phase, they are simply forwarded without seq/ack number translation or book-keeping.

The policy decision tree is depicted in fig. 5. "Is Cookie Correct?" represents re-computing the SYN cookie and comparing it with the acknowledgement number in the client's ACK packet. "To Apply Penalty?" represents checking that the client and its prefix are not generating too much traffic. "Can Replace Current Entry?" represents resolving collisions in the hash table. If the current entry is known to be compliant (*i.e.*, its *conformance* bit is set), then the new entry is dropped. If conformance is still being measured, the new entry is dropped, too. Otherwise, the old entry is evicted and the new entry is inserted in its place.

In all cases, processing is limited to a few hash table lookups, and access to the packet is limited to the information in the header (IP addresses, port numbers, sequence numbers, packet type). Detailed performance evaluation can be found in section 5.

3.4 Evasions and Attacks on the Middlebox

In this section, we focus on the denial of service attacks against the middlebox itself, and on techniques that an attacker may use to evade our defenses.

Exhausting the connection state. To prevent the attacker from filling up the Connection table, we use the Src-Dest table to limit the number of connections from any single host. For protection from botnets, we use source-prefix whitelisting as described in section 3.2.2. In general, resource exhaustion is prevented because the middlebox keeps state only for unspoofed sources that have complied with traffic control measures (*i.e.*, whose network-level behavior is similar to legitimate sources).

Adaptive traffic variation. The attacker may employ an ON/OFF attack pattern. On attack detection, the middlebox is introduced on the data path. As soon as middlebox is introduced, the attacker stops sending attack traffic. All legitimate traffic goes via the middlebox and suffers minor degradation due to triangular routing. After some time interval (the attacker assumes that the middlebox is now removed from data path), he starts sending attack traffic again, and so on. To provide a partial defense against this attack, we avoid rapid introduction and removal of middleboxes. Once the middlebox is introduced, it remains in the data path for some period even after the attack subsided. The duration of this interval is randomized.

Werewolf attack. The attacker starts by behaving legitimately, gets established in the Connection table, complies with congestion control requests, and then starts bombarding the server with attack traffic. We deal with this attack by periodically re-measuring traffic sending rates and source-prefix whitelisting.

Multiple attacks. The attacker can try to overwhelm the dFence infrastructure by launching multiple attacks on

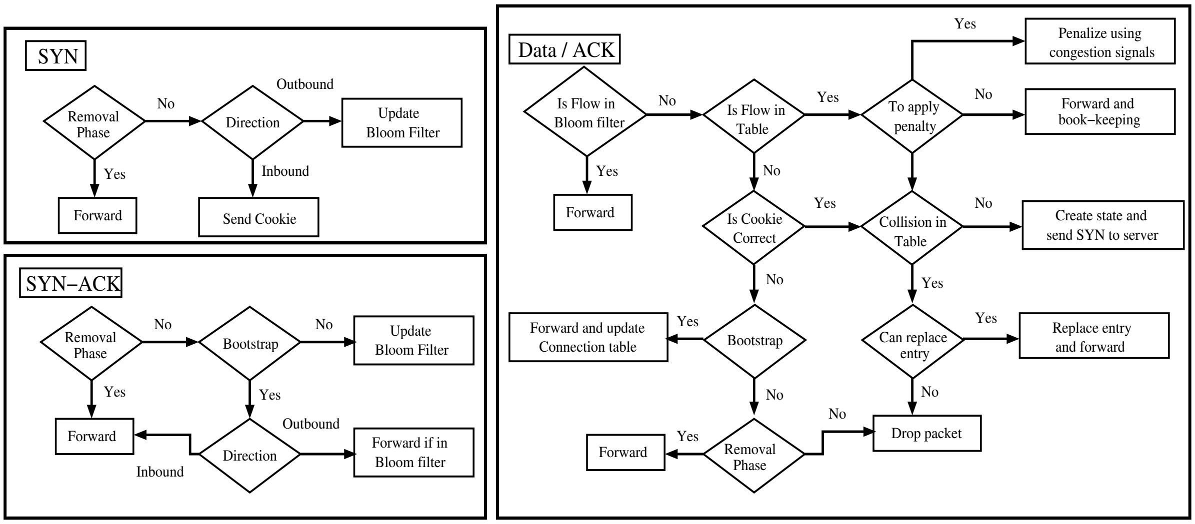


Figure 5: Policy decision tree

several destinations. We employ an adaptive provisioning strategy that scales up the number of middleboxes in the network with the number of attacked destinations (among those who have subscribed for dFence protection).

4 Implementation

Our prototype implementation consists of two components: (i) control-plane traffic interception, and (ii) data-plane attack mitigation. We prototyped the control plane functionality on a general-purpose processor using the extensible open-source router platform called XORP [13]. The anti-DoS data plane functionality is implemented on a special-purpose platform consisting of a Radisys ENP-2611 board, with a 600 MHz Intel® IXP2400 network processor.

The IXP2400 network processor contains one 32-bit XScale controller running Montavista Linux and eight 32-bit RISC cores called micro-engines (MEs). Each ME has a private 4K instruction store, onto which code for different packet processing functions (PPFs) is loaded. The micro-engines share a 16KB on-chip scratch-pad, off-chip SRAM (128MB), and DRAM (1GB).

The complete setup is depicted in Figure 6(a). The control plane uses BGP and IGP to make routing decisions and update the forwarding table. The data packets are handled on the fast path by IXP.

Control plane interception. The middlebox starts its operation after it receives the signal that a DoS attack has been detected. (Our focus in this paper is solely on mitigation rather than detection; dFence is compatible with any existing DoS detection mechanism—see section 6.) As discussed in Section 2.1, the middlebox intercepts traffic to the hosts experiencing the attack by sending iBGP advertisements to all routers within the same AS. Using BGP policy configuration in XORP, the local preference in the advertisements is set higher than the other routers. As a result, all border and intermediate routers make one of the middleboxes their next hop on the routes to the attacked hosts. Note that iBGP advertisements are sent only for the network prefix(es) under attack. To set up tunnels and ACL rules, the middlebox remotely configures the egress router. This is needed to prevent filtered packets from looping back to the middlebox—see Section 2.1.

Data plane mitigation. The attack mitigation policies are implemented on IXP network processors using the Shangri-La framework [19]. Shangri-La provides a flexible high-level programming environment that facilitates rapid development of packet-processing applications. We chose IXP over Click primarily for pragmatic reasons: the IXP multi-processor architecture supports multiple threads and hence provides higher throughput.

We implemented our mitigation policies as an application graph of packet processing functions (PPFs), operating on different packet types (SYN, data, and so on). The PPFs, as shown in Figure 6(b) are mapped to the IXP micro-engines using the Shangri-La run-time system.

Control-data planes interaction. The fast forwarding path on the data plane uses forwarding table entries established by the control plane to put the packets on appropriate output interfaces. We implemented the communication interface between the control plane (on XORP) and data plane (on IXP) using sockets and ioctl() system calls. Communication between a XORP process and a process running on the XScale processor occurs via standard C sockets, and communication between XScale and micro-engines occurs via ioctl() (see Figure 6(a)).

The XORP process sets up the MAC/IP addresses of the interfaces on the IXP data plane, and establishes the mapping between next-hop IP and port numbers. To set up the forwarding table, XORP runs BGP/IGP on control interfaces (on the host processor) and communicates the forwarding table entries to the IXP so that the data plane applications can use the table to forward data packets.

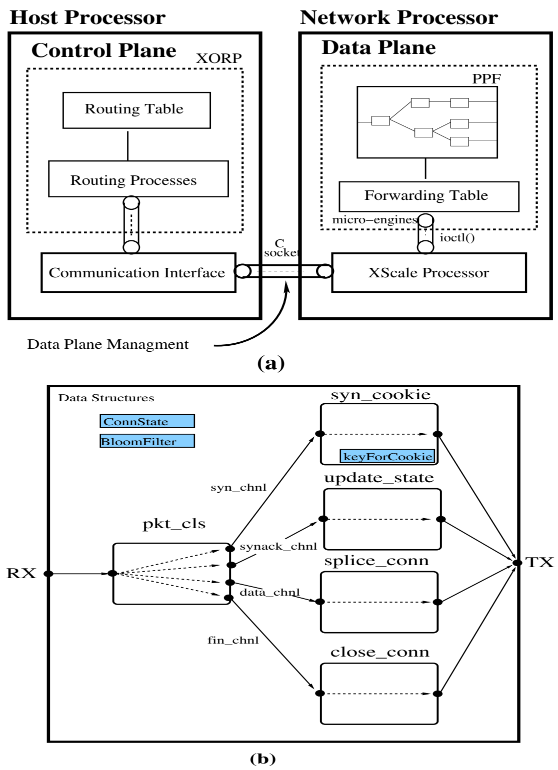


Figure 6: dFence System Implementation. (a) Control plane interception is implemented using XORP on a general-purpose processor. Data plane attack mitigation is implemented on Intel IXP network processors. (b) PPFs for attack mitigation policies.

5 Evaluation

In this section, we present an experimental evaluation of our prototype system. The IXP-based prototype implementation of the middlebox is described in section 4. Attack traffic comprising spoofed SYN packets, data packets, and spoofed ACK/RST/FIN is generated using IXIA packet traffic generator [15]. IXIA has 20 copper ports and two fiber ports. Each fiber port can generate up to 1490 Kilo packets per second, where packet size is 64 bytes.

5.1 Micro benchmarks

To measure throughput and latency of our attack mitigation policies, we directly connect the IXIA fibers ports to two optical ports on the IXP. Traffic generated using IXIA is processed by PPFs on the micro-engines. IXP 2400 has eight micro-engines, two of which are used for receive/transmit modules. We compose the application using four PPFs, each handling a particular packet type: SYN, SYN-ACK, data and FIN/RST. The four PPFs are mapped onto one micro-engine each. The PPF for packet classification is mapped to the same micro-engine as the PPF for FIN/RST. PPFs can be also be mapped to more than one micro-engine, where the code for the PPF is replicated on all the engines.

Packet Type	Packet Processing and Forwarding	Min	Max	Avg
SYN	SYN Cookie and SYN-ACK Generation	39.3	60.4	42.1
	Bloom filter update	25.28	27.06	25.86
	No processing	15.66	17.02	15.94
Inbound Data	Present in Bloom filter	23.24	24.8	23.56
	Present in Connection Table - splice	37.06	40.84	38.61
	Absent in both - forward (removal phase)	31.8	34.06	32.54
Outbound Data	Present in Bloom filter	23.24	25.3	23.56
	Present in Connection Table	37.66	41.64	39.1
	Absent in both - forward (removal phase)	29.52	44.92	30.15
	Absent in both - update (bootstrap phase)	31.5	33.6	32.1

Table 1: Latency benchmarks (in micro-seconds).

Synthetic traffic generated by IXIA consists of 100-byte packets. The maximum input traffic rate attainable in this case is 1041 Kilo packets per second. SYN packets are generated with arbitrary sequence numbers. Since our mitigation policies at the IXP drop packets with invalid sequence/ack numbers, we configure IXIA to automatically insert appropriate numbers into data and FIN packets. To enable testing over longer periods, we disable the interval-based key for generating SYN cookies. Instead, we use a single key that persists over the entire duration of testing using IXIA. This ensures that the data packets with appropriate seq/ack numbers (corresponding to those generated by the middlebox as part of SYN cookie generation) have their flow identifiers in the Connection table and are spliced properly by the IXP.

Latency. Table 1 shows the latency (in micro-seconds) introduced by the middlebox when dealing with different packet types and for different types of processing. Latency includes both processing and packet forwarding. Bloom filter update is performed only for SYN packets from the hosts that are being protected (all such connections are assumed to be legitimate). "Present in Bloom filter" checks the existence of flow ID (IP addresses and ports) in the Bloom filter, and forwards if present (*i.e.*, the packet belongs to an existing server-originated connection). "Present in Connection Table" checks whether the flow ID is present the Connection and, if so, forwards according to the status bits (splicing - seq/ack number translation; pre-existing - connection was classified as legitimate during the bootstrap phase). "Absent in both - forward" applies during the removal phase, when all data packets are simply forwarded. "Absent in both - update" applies during the bootstrap phase: middlebox state is updated for packets received from the protected server by setting the *pre-existing* status bit to *true*.

The latency of updating the Bloom filter (done only during bootstrap phase) is higher than checking the filter. For data packets, checking in the Connection table and splicing (seq/ack number translation + incremental TCP

Packet Type	Packet Processing and Forwarding	1 ME	2 ME	3 ME
SYN	SYN Cookie and SYN-ACK Generation	205	401	467
	Bloom filter update	530	1041	1041
	Forward	1041	1041	1041
Inbound Data	Present in Bloom filter	507	1011	1041
	Present in Connection Table - splice	264	525	781
	Absent in both - forward (removal phase)	326	652	974
Outbound Data	Present in Bloom filter	507	1011	1041
	Present in Connection Table	259	515	766
	Absent in both - forward (removal phase)	318	637	1029
	Absent in both - update (bootstrap phase)	326	653	951

Table 2: Throughput benchmarks in Kilo Packets Per Second (Kpps). Maximum input rate from IXIA (one fiber port) is 1041 Kpps with packet size = 100 bytes.

checksum computation) is more expensive than checking the Bloom filter, updating, or simple forwarding.

Throughput. Table 2 presents our throughput benchmarks. Throughput scales linearly as more micro-engines are allocated to the PPFs for all packet types and processing functionalities, except for SYN cookie generation. For the latter, maximum throughput supported by a single IXP is 467 Kpps.

5.2 End-to-end benchmarks

For end-to-end measurements, our server is a 1 GHz Intel P-III processor with 256 MB RAM, 256 KB cache, running an Apache Web Server on Linux 2.4.20 kernel. Legitimate traffic is generated using the httperf [23] tool which issues HTTP requests to the Web server. Both the client and the server are connected to a Gigabit Ethernet switch. Spoofed attack traffic is generated using IXIA, which is connected to the fiber optical port of the switch. All traffic goes via a routing element running XORP. For our prototype, we do not include attack detection and use a trigger to install the middlebox on the data path.

Our evaluation metrics are *connection time*, measured using httperf, and *max TCP throughput* attainable between a legitimate client and the server, measured using iperf [14].

Latency. In Figure 7(a), X axis represents the attack rate in Kilo packets per second (100-byte packets), Y-axis represents connection time in milliseconds. For server content, we used `www.amazon.com` homepage (copied on April 17, 2006). Its size is 166 KB.

For one legitimate connection, no attack traffic and no middlebox on the data path, connection time is 16.4 ms. With the middlebox on the data path, but still no attack, connection time increases to 16.5 ms. For ten concurrent connections, total connection time increases from 121.1 ms to 121.5 ms.

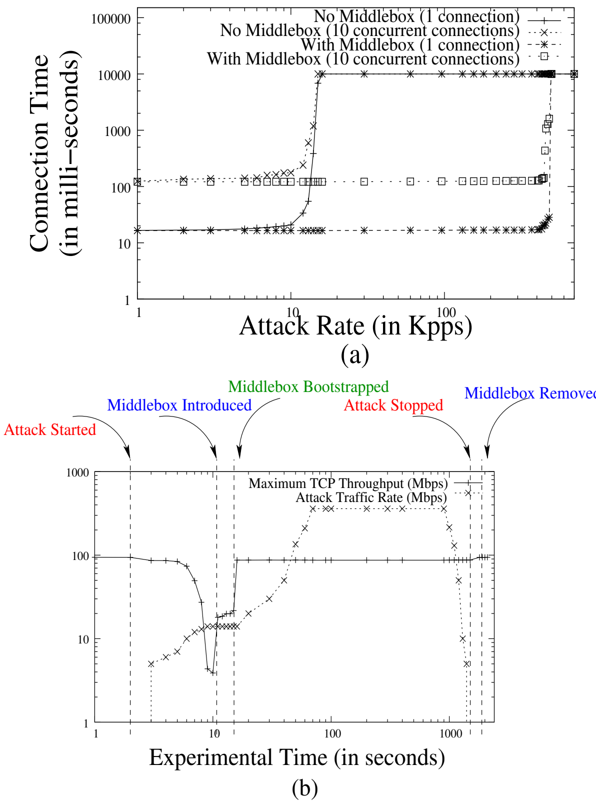


Figure 7: (a) End-to-end latency for one and ten concurrent HTTP connections to a Web server. Attack traffic rate is increased up to 490 Kpps (100-byte packets); (b) End-to-end maximum TCP throughput. Attack traffic rate, and TCP throughput are in Mbps.

As seen from Figure 7(a), connection time with no middlebox on the data path increases as attack traffic rate grows. After around 14 Kpps, the Web server can no longer handle the traffic and httperf client times out. The timeout interval is set to be 10 seconds. At this moment, the server dies. With the middlebox on the data path, connection time for legitimate clients remains constant even as attack rate increases all the way to 450 Kpps.

Throughput. Fig. 7(b) shows end-to-end performance (measured using iperf) over time as the server is attacked, middlebox enters dynamically into the data path, bootstraps, filters out attack traffic, and, after the attack subsides, is removed from the data path.

Before the attack starts, maximum TCP throughput between client and server is 94.3 Mbps. As the attack begins, it drops to 3.88 Mbps. After $t = 10$ seconds, the middlebox is dynamically introduced on the data path. During the 6-second bootstrap phase, the middlebox establishes state for ongoing connections, and throughput slowly increases to 21.7 Mbps (the increase is due to dropping of spoofed SYN requests - these packets do not get to the server, because the TCP handshake between the attacker and the middlebox is not completed). All data packets, whether spoofed or legitimate, are forwarded towards the server during the bootstrap phase (note, however, that the attack traffic rate stays below 14 Kpps). At $t = 16$, the middlebox enters its active mode, and starts aggressively profiling and filtering traffic. All spoofed traffic is dropped in this phase. Throughput now increased to 87.3 Mbps. At $t = 1500$, the attack stops, and the middlebox remains on the data path for the next 300 seconds. This interval (pre-determined) is used to time out the state for connections that were established via the middlebox during the active phase. At $t = 1800$, throughput returns to the normal (no attack, no middlebox) 94.3 Mbps level.

6 Related Work

Defenses against denial of service have been a subject of very active research, and the survey in this section is necessarily incomplete. Unlike previously proposed network-based defenses, dFence is completely transparent to the existing Internet infrastructure. Unlike proxy-based solutions, dFence uses novel dynamic introduction mechanisms to provide *on-demand* protection only when needed. dFence middleboxes can be quickly re-deployed to protect a different subset of end hosts without any modifications.

Network-based mitigation. Defenses based on secure overlays [2, 17] assume that all packets enter the network through the overlay's access points. The overlay checks each packet's legitimacy and filters out attack traffic. This method requires that the destinations' true IP addresses remain secret, and is thus difficult to combine with the existing Internet infrastructure. Similarly, Firebreak [11] assumes that the attacker does not know the targets' IP addresses, and that packets are tunnelled to the destinations by proxies deployed at edge routers. This requires software modification at legacy routers.

Defenses based on capabilities such as SIFF [36] and TVA [37] require that (i) destinations issue unforgeable tokens to legitimate sources, and (ii) routers filter out packets that do not carry these tokens. Both router and server software must be modified to support capabilities, and servers must be able to differentiate benign and malicious traffic. Flow Cookies [6] use the timestamp field in packets to insert cookies, and require server modifications to differentiate benign and malicious flows.

Pushback [21] rate-limits flows responsible for traffic congestion, and pushes filters upstream towards the sources of these flows. Router software must be modified. Rate-limiting is a coarse technique that does not differentiate between benign and malicious traffic, and may thus cause high collateral damage.

Cisco Guard [8] is a commercial product that dynamically redirects traffic to "cleaning centers" within the network. Traffic interception is not bi-directional; only traffic from client to server is intercepted. Cisco Guard applies several stateless filtering policies, and uses rate-limiting to reduce traffic volume (which may potentially cause high collateral damage). In contrast, our scheme intercepts both directions of traffic and supports both stateless and stateful policies to enable better differentiation between benign and malicious traffic.

Several designs for re-engineering the Internet infrastructure have resistance to denial of service attacks among their objectives [30, 31]. With indirection as the first-class principle of packet routing, these networks can easily reroute attack traffic to filtering devices by changing the mappings between identifiers and hosts. The scheme proposed in this paper is incomparable, because our goal is a solution that is fully compatible with and transparent to the existing Internet infrastructure.

Other network-based defenses, all requiring router modification, include route-based packet filtering [25], statistical analysis of incoming packets [18] and router throttles [38]. An evaluation of router-based defense systems can be found in [34].

Victim- and source-based mitigation. These defenses are deployed either at the servers, or at the ingress routers, and thus necessarily require substantial modifications to the existing software base. Server-based solutions also tend to be ineffective against last-mile bandwidth flooding attacks.

Kill-Bots [16] uses client legitimacy tests such as reverse Turing tests to differentiate between benign and malicious requests. In [32], victim servers encourage legitimate clients to "crowd out" malicious flows by sending higher volumes of traffic. In Pi [35], routers insert path identifiers into unused spaces within IP packet headers; servers then drop packets arriving on known attack paths. This requires modifications to both routers and servers, and may cause collateral damage if a legitimate source shares the route with an attacker. D-WARD [22] uses anomaly detection and compliance with traffic management measures to differentiate benign and malicious flows. Malicious flows are then blocked or rate-limited at source routers. Deployment requires wide-scale modification of router software. Ingress filtering [10] is limited to spoofing attacks, and also requires router modification.

Many methods have been proposed for *detecting* denial of service activity [12, 33, 28] and tracing back the sources of the attack [27, 29]. Our focus in this paper is on transparent, scalable *mitigation* rather than detection, and our solution is compatible with most proposed detection and traceback mechanisms.

7 Conclusions and Future Work

We described the design and prototype implementation of dFence, a novel network-based system for transparently mitigating denial of service attacks. The main advantages of the dFence middleboxes are their compatibility with the existing Internet infrastructure—they are introduced into the network using standard routing mechanisms, and their operation is completely transparent to the protected end hosts—and their ability to support a broad range of effective anti-DoS techniques. Control over both directions of TCP connections and efficient data structures for managing partial connection state enable several new defenses against denial of service, and make possible on-demand deployment of defenses in the middle of the network. Our experimental evaluation demonstrates that dFence provides effective protection against distributed DoS attacks at a minimal performance cost. Moreover, there is no impact whatsoever on traffic to servers that are not experiencing DoS attacks.

Future work includes investigation of mechanisms for configuration and management of dFence middleboxes, as well as design and implementation of an extensible scripting language for rapid development of new anti-DoS policies. Another research objective is a better understanding of *adaptive* attacker behavior and designing defenses against attackers who are aware of the anti-DoS middleboxes and deliberately craft their attack patterns to evade mitigation policies. This includes game-theoretic modeling of adversarial interaction between the middleboxes and the attackers. Finally, we would like to extend dFence to environments with multiple ISPs.

8 Acknowledgements

We are grateful to the anonymous NSDI reviewers and our shepherd Eugene Ng for insightful comments. Their suggestions have significantly improved our paper. We thank Tae Won Cho and Upendra Shevade for helpful discussions. Finally, we thank National Science Foundation for sponsoring the research under grant CNS-0546720.

References

[1] S. Agarwal, T. Dawson, and C. Tryfonas. DDoS mitigation via regional cleaning centers. Sprint ATL Research Report RR04-ATL-013177, January 2004.

[2] D. Andersen. Mayday: Distributed filtering for Internet services. In *Proc. USITS*, 2003.

[3] ANML. DDoS attack tools. `http://anml.iu.edu/ddos/tools.html`, 2001.

[4] D. Bernstein. SYN cookies. `http://cr.yp.to/syncookies.html`, 1996.

[5] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4), 2004.

[6] M. Casado, A. Akella, P. Cao, N. Provos, and S. Shenker. Cookies along trust-boundaries (CAT): Accurate and deployable flood protection. In *Proc. SRUTI*, 2006.

[7] Cisco. Policy-based routing. `http://www.cisco.com/warp/public/732/Tech/plicy_wp.htm`, 1996.

[8] Cisco. Cisco Guard DDoS mitigation appliances. `http://www.cisco.com/en/US/products/ps5888/`, 2007.

[9] D. Dagon, C. Zou, and W. Lee. Modeling botnet propagation using time zones. In *Proc. NDSS*, 2006.

[10] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. `http://www.faqs.org/rfcs/rfc2827.html`, 2000.

[11] P. Francis. Firebreak: An IP perimeter defense architecture. `http://www.cs.cornell.edu/People/francis/hotnets-firebreak-v7.pdf`, 2004.

[12] T. Gil and M. Poletto. MULTOPS: A data-structure for bandwidth attack detection. In *Proc. USENIX Security*, 2001.

[13] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing extensible IP router software. In *Proc. NSDI*, 2005.

[14] Iperf. The TCP/UDP bandwidth measurement tool. `http://dast.nlanr.net/Projects/Iperf/`, 2003.

[15] IXIA. `http://www.ixiacom.com`, 2006.

[16] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-Sale: Surviving organized DDoS attacks that mimic flash crowds. In *Proc. NSDI*, 2005.

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

[18] Y. Kim, W. Lau, M. Chuah, and J. Chao. PacketScore: Statistics-based overload control against distributed denial-of-service attacks. In *Proc. INFOCOM*, 2004.

[19] R. Kokku, U. Shevade, N. Shah, A. Mahimkar, T. Cho, and H. Vin. Processor scheduler for multi-service routers. In *Proc. RTSS*, 2006.

[20] J. Lemon. Resisting SYN flood DoS attacks with a SYN cache. In *Proc. BSDCon*, 2002.

[21] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *CCR*, 32(3), 2002.

[22] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. In *Proc. ICNP*, 2002.

[23] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. *Performance Evaluation Review*, 26(3), 1998.

[24] D. Pappalardo and E. Messmer. Extortion via DDoS on the rise. Network World, May 16 2005.

[25] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets. In *Proc. SIGCOMM*, 2001.

[26] Prolexic. The Prolexic Zombie Report. `http://www.prolexic.com/zr/`, 2007.

[27] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Network support for IP traceback. *IEEE/ACM Trans. Netw.*, 9(3), 2001.

[28] V. Sekar, N. Duffield, K. van der Merwe, O. Spatscheck, and H. Zhang. LADS: Large-scale automated DDoS detection system. In *Proc. USENIX*, 2006.

[29] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, S. Kent, and W. Strayer. Hash-based IP traceback. In *Proc. SIGCOMM*, 2001.

[30] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proc. SIGCOMM*, 2002.

[31] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *Proc. OSDI*, 2004.

[32] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *Proc. SIGCOMM*, 2006.

[33] H. Wang, D. Zhang, and K. Shin. Detecting SYN flooding attacks. In *Proc. INFOCOM*, 2002.

[34] Y. Xu and R. Guerin. On the robustness of router-based denial-of-service (DoS) defense systems. *CCR*, 35(3), 2005.

[35] A. Yaar, A. Perrig, and D. Song. Pi: A path identification mechanism to defend against DDoS attacks. In *Proc. IEEE S&P*, 2003.

[36] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks. In *Proc. IEEE S&P*, 2004.

[37] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *Proc. SIGCOMM*, 2005.

[38] D. Yau, J. Lui, F. Liang, and Y. Yam. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. *IEEE/ACM Trans. Netw.*, 13(1), 2005.