PROJECT REPORT ON
DEFENSE AGAINST DISTRIBUTED DENIAL OF SERVICE ATTACKS

SUBMITTED IN COMPLETE FULFILLMENT OF THE REQUIREMENT FOR
THE DEGREE OF
BACHELOR OF ENGINEERING IN INFORMATION TECHNOLOGY

BY
HARSHA VARDHAN ACHREKAR
JAIME GANDHI
JITEN VITHLANI
MAMTA CHIRMADE
VIBHA SINGHAL

UNDER THE GUIDANCE OF
MRS. NUPUR GIRI

VIVEKANAND EDUCATION SOCIETY’S INSTITUTE OF TECHNOLOGY
UNIVERSITY OF MUMBAI

*2003-2004*
ACKNOWLEDGEMENT

We would like to take this opportunity to express our sincere gratitude to our Project Guide Mrs. Nupur Giri for her support & guidance. It is her support and enthusiasm that this small idea has bloomed into an international research project and has been implemented. We would also take this opportunity to profusely thank the Computer Department, Vivekanand Institute of Technology for the support and co-operation. Last but not the least; we would like to thank the faculty & staff of the Information Technology Department at Vivekanand Education Society’s Institute of Technology.
DEFENSE AGAINST DISTRIBUTED DENIAL OF SERVICE ATTACKS

1. Introduction
   1

2. Problem Definition and Analysis
   2
   2.1 The Essence of a Distributed Denial of Service Attack
       3
   2.2 How is DDoS Executed
       5

3. Conceptual Prerequisite
   7
   3.1 Network Ingress Filtering
       8
       An Example of an Attack
       Restricting Forged Traffic
   3.2 Egress Filtering
       11
   3.3 CLICK Router Architecture
       12
       Click as Packet Processor
       Click Design Principles
       Click Components
   3.4 MULTOPS
       16
       MULTOPS Heuristics
       Data Structure
   3.5 Solution Requirements
       21

4. Component-Level Design
   23
   4.1 MULTOPS Component
       25
       PUSH Function
       PULL Function
       Update function
       Make Counter Function
       FOLD Function
   4.2 DIFFERENTIATOR Component
       30
   4.3 Router Logic
       31

5. System Architecture
   33
   5.1 Implementation Set-up
       34
   5.2 System Specification
       35
DEFENSE AGAINST DISTRIBUTED DENIAL OF SERVICE ATTACKS

6. Implementation (source code)  
   6.1 Differentiator.hh  
   6.2 Differentiator.cc  
   6.3 ICMP.click  
   6.4 Attack.c  
   6.5 Router Script  
   6.6 Createlog.c  

APPENDIX (Implementation Prerequisites)  
   A. Upgrading the default Kernel  
   B. Installing CLICK in Kernel mode  
   C. Compiling and Patching new element in CLICK
1. Introduction

In the recent past, new classes of security threats have emerged: one, which does not target the integrity of resources, but rather their very availability. All threats under the new class are collectively known as DENIAL OF SERVICE (DoS) ATTACKS. DoS do not associate with the actual contents, they just concentrate on preventing the data from reaching its actual destination, thus effectively making the resource or service unavailable by exploiting either flaws in popular internet protocols or the very functioning of internet.

A key problem in detecting denial of service attacks is that the source address of the packets is spoofed. This ensures that the compromised machines remain undetected and thereby can be used for other attacks. If the source of the attack is kept constant (even if it is spoofed), it is possible to block that particular address and recover from the attack. However, the attack now takes a new form by being distributed (DDoS). In this form, a number of compromised systems all over the world are used in a synchronized manner to attack a particular server. By distributing the attack, the intensity near the source is lessened and is therefore not detected there. Meanwhile, the concentrated effect at the victim is sufficient to overload networks and systems and thus deny service. This latest evolution in DoS has received much publicity, but some of the most important aspects have not yet been explored. DDoS isn’t simply about multiplication of attack sources, it brings about issues of path diversity, obscurity, invisibility, and demoralization of the victim.

What we concluded is that flow rather than the content holds greater relevance as far as developing a generic solution goes. Hence we concentrate on a solution that utilizes the traffic pattern characteristics to handle the immense traffic volumes generated by DDoS attacks, irrespective of the exploit or the technique used.
2. PROBLEM DEFINITION AND ANALYSIS
2. Problem Definition

The recent occurrences of DDoS attacks make it an important issue to deal with. Of the various technologies available for its prevention, network ingress filtering and egress filtering are implementable, effective and reliable methods. We thus implement network ingress and egress filtering with the help of CLICK Router configuration and MULTOPS data structure to avoid DDoS attacks.

2.1 The Essence of a Distributed Denial of Service (DDoS) Attack:

A “denial-of-service” attack is characterized by an explicit attempt by attackers to prevent legitimate users of a service from using that service. Examples include attempts to “flood” a network, thereby preventing legitimate network traffic attempts to disrupt connections between two machines, thereby preventing access to a service attempts to prevent a particular individual from accessing a service attempts to disrupt service to a specific system or person Denial-of-service attacks come in a variety of forms and aim at a variety of services. There are three basic types of attack: consumption of scarce, limited, or non-renewable resources by sending illegitimate traffic there by denying service to the legitimate users. Destruction or alteration of configuration information physical destruction or alteration of network components

The basic intent of a DoS attack is either to overwhelm the resources allocated by a networked device to a particular service in order to prevent its use, or to crash a target device or system. DoS attacks can achieve this goal by either taking advantage of known bugs in particular OS versions, or by taking advantage of ‘loopholes’ designed into the fabric of a standard such as TCP/IP. DoS attacks can be effective by focusing on routers, switches, or servers.
As an example, the strength of the routers, networks, applications and servers that comprise the infrastructure of the Internet, is their ability to handle a large number of requests for connection simultaneously. The weakness in the strength is in how they do so. Each time a client requests connection to a network service or application, a corresponding server sets aside a portion of its resource to handle the connection. By flooding a target server with connection requests, the finite resources allocated to a specific service can be overwhelmed. The result is a Denial of Service to valid connection requests, system errors, and possible system crashes.

DoS attacks can generally be classified as either a Flood Attack or a Malformed (or crafted) Packet Attack and that where attacks originate simultaneously from several compromised sources that these can be classified as Distributed DoS attacks.

Fundamental to the IP protocol every packet has a source and destination address field that is used to determine the originating and destination end points. The process of forwarding these packets by intermediate routers partly relies on the destination field; the source address will only be used when a response to the packet is required. This makes the implementation of DDoS flooding attacks easy to accomplish because fake or “spoofed” source addresses can be used, and packets will generally be forwarded unchallenged to the specified destination. This allows a DoS or DDoS attack to be carried out from any location and with total anonymity.

If an attack is underway from a single address then it is possible to arrange for a “block” of the offending source IP address at the ISP or the border router. However, when a DDoS attack occurs the problem is not as easy to resolve because packets appear to be coming from hundreds or even thousands of different hosts, there is absolutely no point trying to implement temporary Access Control Lists on routing devices or modify the border Firewall rulebase, it is too late – you are left at the mercy of the attack under way.
The types of attack can also take the form of a single “one shot” crafted packet originating from a single host to thousands of packets per second originating simultaneously from multiple hosts.

### 2.2 How is DDoS Executed:

The Distributed Denial of Service (DDoS) attack works as follows. The attacker uses widely available hacker tools to probe unsuspecting networks of computers for security weaknesses (trust us – you have those). These tools often scan for and then ‘piggyback’ on network sessions between key resources in your network and users authorized to control or reconfigure them.

Once a suspect is located, the attacker uses this authorization level to insert a ‘master’ DDoS program on a key system within your network. The ‘master’ is then used to insert ‘slave’ programs on other computers within the same network. These slaves contain one or many common DoS programs. A network of master/slave DoS programs then exists that will lie dormant awaiting a wake up call to attack a specific target system. The hacker then moves on to find additional prospective master/slave networks.

The normal DDoS attack architecture works upon the basis that the required hosts to launch the attack from have already been identified and compromised via Trojans or “backdoors. In a DDoS scenario the Intruder (also called the Attacker or Client) issues control traffic to the Master (also called the Handler) which, in turn then issues commands to the Daemon (also called an Agent, Broadcast program or Zombie). The Daemons that are at the end of this command chain finally initiate the attack traffic against the Victim. This distributed architecture increases the attack capability many times over and allows the Intruder the means to remain undetected as shown in Figure 2 below.
Defense Against Distributed Denial of Service Attacks

![Diagram of defense against Distributed Denial of Service Attacks.]

Figure 2
3. CONCEPTUAL PREREQUISITE
3.1 NETWORK INGRESS FILTERING:

DDoS attacks, which have employed forged source addresses, have proven to be a troublesome issue for Internet Service Providers. Ingress filtering is one method to reduce DDoS attacks which use forged IP addresses to be propagated from ‘behind’ an Internet Service Provider’s (ISP) aggregation point. Ingress filtering applies to traffic received at the router from the customer. While ingress traffic filtering reduces the success of source address spoofing, it does not preclude an attacker using a forged source address of another host within the permitted prefix filter range. It does, however, ensure that when an attack of this nature does indeed occur, a network administrator can be sure that the attack is actually originating from within the known prefixes that are being advertised. This simplifies tracking down the culprit, and at worst, an administrator can block a range of source addresses until the problem is resolved. All providers of Internet connectivity are urged to implement ingress filtering to prohibit attackers from using forged source addresses that do not reside within a range of legitimately advertised prefixes.

3.1.1 An Example of an attack:

Of all the attacks causing denial of service, the TCP-SYN Flooding attack is most common. A simplified diagram of the TCP SYN flooding problem is depicted below:
Assume:

- The “host” is the targeted machine.
- The attacker resides within the “valid” prefix, 9.0.0.0/8.
- The attacker launches the attack using randomly changing source addresses; in this example, the source addresses are depicted as from within, which are not generally present in the global Internet routing tables, and therefore, unreachable. However, any unreachable prefix could be used to perpetrate this attack method.

Also worthy of mention is a case wherein the source address is forged to appear to have originated from within another legitimate network which appears in the global routing table(s). For example, an attacker using a valid network address could wreak havoc by making the attack appear to come from an organization which did not, in fact, originate the attack and was completely innocent. In such cases, the administrator of a system under attack may be inclined to filter all traffic coming from the apparent attack source.
Adding such a filter would then result in a denial of service to legitimate, non-hostile end-systems. In this case, the administrator of the system under attack unwittingly becomes an accomplice of the attacker.

### 3.1.2 Restricting forged traffic:

The problems encountered with this type of attack are numerous, and involve shortcomings in host software implementations, routing methodologies, and the TCP/IP protocols themselves. However, by restricting transit traffic which originates from a downstream network to known, and intentionally advertised, prefix(es), the problem of source address spoofing can be virtually eliminated in this attack scenario.

```
11.0.0.0/8
/  
router 1
/  
/  
/  9.0.0.0/8
ISP <- ISP <- ISP <- ISP <- router <- attacker
A B C D 2
/  
/  
/  
router 3
/  
12.0.0.0/8
```

In the example above, the attacker resides within 9.0.0.0/8, which is provided Internet connectivity by ISP ‘D’. An input traffic filter on the ingress (input) link of “router 2”, which provides connectivity to the attacker’s network, restricts traffic to allow only traffic...
originating from source addresses within the 9.0.0.0/8 prefix, and prohibits an attacker from using “invalid” source addresses which reside outside of this prefix range.

In other words, the ingress filter on “router 2” above would check:

\[
\begin{align*}
&\text{IF} & \text{packet’s source address from within 9.0.0.0/8} \\
&\text{THEN} & \text{forward as appropriate} \\
&\text{IF} & \text{packet’s source address is anything else} \\
&\text{THEN} & \text{deny packet}
\end{align*}
\]

Network administrators should log information on packets which are dropped. This then provides a basis for monitoring any suspicious activity.

### 3.2 EGRESS FILTERING:

In customer networks, it is customary to create inbound access rules to control what traffic is allowed in from the Internet. All too often however, many administrators pay little attention to what is allowed out of their network. In other words, egress filtering, or the filtering of outbound traffic is not being performed. This can make customer networks an excellent haven for DDoS attacks. The best way to ensure that only assigned IP address space leaves customer networks is to setup an outbound filter on the customers egress router. Besides ensuring that spoofing attacks cannot be launched from a customers’ network, it’s also a great way to insure that private addressing is not leaked out.
3.3 CLICK ROUTER ARCHITECTURE:

Click is new software architecture for building flexible and configurable routers. A Click router is assembled from packet processing modules called elements. Individual elements implement simple router functions like packet classification, queuing, scheduling, and interfacing with network devices. A router configuration is a directed graph with elements at the vertices; packets flow along the edges of the graph. Configurations are written in a declarative language that supports user-defined abstractions. This language is both readable by humans and easily manipulated by tools.

Due to Click’s architecture and language, Click router configurations are modular and easy to extend. A standards-compliant Click IP router has sixteen elements on its forwarding path.

3.3.1 Click As Packet Processor:

Click is an extensible toolkit for writing packet processors. A packet-processing network consists of first order routers and hosts. Hosts use packets as a means to an end; they are mostly concerned with providing communication abstractions to applications. Routers, however, are pure packet processors. They are interested only in packets, which they route from place to place based on packet header information. Routing was the first packet processing application on the Internet, but many others have come to light as the network has matured. Firewalls limit access to a protected network, often by dropping inappropriate packets. Network address translators allow a large set of machines to share a single public IP address; they work by rewriting packet headers, and occasionally some data. Load balancers send packets to one of a set of servers, dynamically choosing a server based on load or some application characteristic. Other packet processors enforce quality of service policies, monitor traffic for planning purposes, detect hacker attacks or inappropriate use of resources, and support mobile hosts.
The essential characteristic shared by all packet processors is the motion of packets. Packets arrive on one network interface, travel through the packet processor’s forwarding path, and are emitted on another network interface. Contrast this with hosts, where packets lose their identity after they arrive: only data is transferred to the application. In packet processors, packets move horizontally between peers, not vertically between application layers.

Click routers are built from fine-grained software components called elements. To build a router configuration, the user chooses a collection of elements and connects them into a directed graph. The graph’s edges represent possible paths for packet transfer. This layerless design was motivated by the peer-to-peer nature of packet processing.

![Click components](image_url)

**Figure 3.1: An illustration of click components.**

The Click architecture is centered on the element. Each element is a software component representing a unit of router processing. Elements perform conceptually simple computations, such as decrementing an IP packet’s time-to-live field, rather than large, complex computations, such as IP routing. They generally examine or modify packets in some way; packets, naturally, are the particles of network data that routers exist to process. At run time, elements pass packets to one another over links called connections. Each connection represents a possible path for packet transfer. Click router configurations are directed graphs of elements with connections as the edges. Router configurations, in turn, run in the context of some driver, either at user level or in the Linux kernel.

Figure 1 shows some elements connected together into a simple router configuration. Elements appear as boxes; connections appear as arrows connecting the boxes together. Other features of the diagram are described in later sections. Packets pass from element to element along
the arrows (connections). This router’s elements read packets from the network (FromDevice(eth0)), count them (Counter), and finally throw them away (Discard).

### 3.3.2 Click Design Principles:
Click’s design began with these principles:

1. **One rich, flexible abstraction.**
   Click’s single component abstraction is the element. Router configurations consist of elements connected together; there are no other component-level abstractions.

2. **Configuration language.**
   Click router design divides naturally into two phases. In the first phase, users write element classes, which are configuration independent. In the second, users design a particular router configuration by choosing a set of elements and the connections between them.

3. **Avoid restrictions.**
   The Click system guides users to create modular router and element designs by making modularity easy to achieve, but it does not prevent bad designs or restrict user flexibility.

### 3.3.3 Click Components:

The element is the most important user-visible abstraction in Click. Every property of a router configuration is specified either through the choice of elements or through their arrangement. Device handling, routing table lookups, queuing, counting, and so forth are all implemented by elements. Inside a running router, each element is a C++ object that may maintain private state.

Elements have five important properties: element class, ports, configuration strings, method interfaces, and handlers.
**Element class:** An element’s class specifies that element’s data layout and behavior.

**Ports:** Each element can have any number of input and output ports. Every connection links an output port on one element to an input port on another. Different ports may have different roles; every port that is provided must be used by at least one connection, or the configuration is in error.

**Configuration string:** The optional configuration string contains additional arguments passed to the element at router initialization time. For many element classes, configuration strings define per-element state and fine-tune element behavior, much as constructor arguments do for objects. Lexically, a configuration string is a list of arguments separated by commas. Most configuration arguments fit into one of a small set of data types.

**Method interfaces:** Each element exports methods that other elements may access. This set of methods is grouped into method interfaces. Every element supports at least the base method interface, which contains, for example, methods for transferring packets. Elements can define and implement arbitrary other interfaces on top of this.

**Handlers:** Handlers are methods that are exported to the user, rather than to other elements in the router configuration. They support simple, text-based read/write semantics, as opposed to fully general method call semantics.

Elements naturally fall into categories distinguished by their input and output port characteristics. Packet sources spontaneously generate packets, either by reading them from the network (FromDevice), reading them from a dump file (From-Dump), creating them from specified data (InfiniteSource, RatedSource), or creating them from random data (RandomSource). They have one output and no inputs. Packet sinks remove packets from the system, either by dropping them (Discard, TimedSink), sending them to the network (ToDevice), writing their contents to a dump file (ToDump), or sending them to the Linux networking stack (ToLinux). They have one input and no outputs. Packet modifiers change
packet data. They have one input and one or two outputs. Packets arriving on the input are modified and then emitted on the first output. The second output, if present, is for erroneous packets. The input and the first output may be agnostic, but the second output is always push.

Packet checkers keep statistics about packets (Counter) or check them for validity (CheckLength, CheckIPHeader). Their ports resemble those of packet modifiers.

Routing elements choose where incoming packets should go based on a packet-independent switching algorithm (Switch, RoundRobinSwitch), general characteristics of packet flow (Meter, PacketMeter, RatedSplitter), or an examination of packet contents (Classifier, HashSwitch, LookupIPRoute). They have one push input and two or more push outputs.

Storage elements (Queue, FrontDropQueue) store packets in memory for later use, yielding up stored packets upon receiving pull requests. They have one push input and one pull output. Scheduling elements (RoundRobinSched, PrioSched) choose packets from one of several possible packet sources. They have one pull output and two or more pull inputs.

Information elements implement language extensions (AddressInfo, ScheduleInfo) or interact with the configuration out-of-band (ControlSocket). They have no inputs or outputs.

### 3.4 MULTOPS:

We propose a heuristic and a data-structure that network devices (such as routers) can use to detect (and eliminate) such attacks. With our method, each network device maintains a data-structure, MULTOPS, that monitors certain traffic characteristics. MULTOPS (MUlti-Level Tree for Online Packet Statistics) is a tree of nodes that contains packet rate statistics for subnet prefixes at different aggregation levels. The tree expands and contracts within a fixed memory budget. A network device using MULTOPS detects ongoing bandwidth
attacks by the significant, disproportional difference between packet rates going to and coming from the victim or the attacker.

A bandwidth attack should be detected close to the attacker rather than close to the victim so that malicious packets can be stopped before they can cause any harm.

MULTOPS is a tree of nodes that contains packet rate statistics for subnet prefixes at different aggregation levels. It dynamically adapts its shape to reflect changes in packet rates, and avoid (maliciously intended) memory exhaustion.

3.4.1 MULTOPS heuristic:

Packets are defined to be malicious (and, thus, may be dropped) if they are destined for a host or subnet from which too few packets are coming back. This heuristic is based on the assumptions that most Internet traffic consists of packet flows, and during normal operations, the rate of packets in a flow going from A to B is proportional to the packet rate going from B to A. Thus, during normal operations on the Internet, the packet rate of traffic going in one direction is proportional to the packet rate of traffic going in the opposite direction. If not, something must be wrong.

This heuristic appears to hold broadly. TCP, the protocol mainly used on the Internet, acknowledges every single—or every k—received packets by sending back a packet, and, therefore, has proportional packet flows.

The following example illustrates the heuristic. If machine A is sending legitimate TCP packets to machine B, but B is suffering under a bandwidth attack, then A's packets will not reach B. Even if some of A's packets reach B, then B’s packets may not reach A because of the overloaded links and routers. In reaction to the absence of B’s packets, A will automatically decrease the sending rate and, eventually, stop sending packets to B altogether. If, on the other hand, A is an attacker that blasts (any type of) packets at B, a MULTOPS-equipped router routing A’s packets to B will detect the disproportional packet rates between them.
and could decide to drop packets going to B. Consequently, B will not have to cope with A’s packets.

Let $R(P)$ be the ratio between the packet rate going to and coming from addresses with prefix $P$. Under normal circumstances, $R$ is close to some constant $k$ for all $P$, i.e., packet rates are proportional for all prefixes. If $R$ drops below $R_{\text{min}}$ or exceeds $R_{\text{max}}$, then a (host in) subnet with prefix $P$ is either under attack or a subnet with prefix $P$ harbors an attacker.

MULTOPS collects packet rates to and from address prefixes so that, given a certain $P$, $R(P)$ can be calculated. Packets may be dropped if they are destined for a host or subnet from which disproportionately fewer packets are coming back, i.e., if $R(P)$ is not between $R_{\text{min}}$ and $R_{\text{max}}$. The sensitivity of MULTOPS can be tuned by changing the values of $R_{\text{min}}$ and $R_{\text{max}}$.

### 3.4.2 Data structure:

![Diagram of MULTOPS](image)

**Figure 3.2: MULTOPS**

MULTOPS is organized as a 4-level 256-ary tree to conveniently cover the entire IPv4 address space. Each node in the tree is a table consisting of 256 records, each of which consists of 3 fields: 2 rates—to-rate and from-rate—and 1 pointer potentially pointing to a node in the next
level of the tree. A table stores all packet rates to and from IP addresses with a common 0-bit, 8-bit, 16-bit, or 24-bit prefix, depending on the level of the tree. Deeper levels of the tree contain packet rates for addresses with a longer prefix. Thus, the root node contains the aggregate packet rates to and from address 0.*.*.*, 1.*.*.*, 2.*.*.*, etc. The 90th record in the root node, for example, contains the packet rates to and from addresses with 8-bit prefix 89, and a pointer to a node that keeps tracks of the aggregate packet rates to and from addresses with that prefix, i.e., 89.0.*.*, 89.1.*.*, 89.2.*.*, etc. The sum of all 256 to-rates and the sum of all 256 from-rates in a node are equal to the to-rate and the from-rate in the parent record of that node. Figure 1 shows a sample MULTOPS.

When the packet rate to or from a subnet reaches a certain threshold, a new subnode is created on the fly to keep track of more fine-grained packet rates, potentially down to per-IP address packet rates. For example, if the aggregate packet rate to or from subnet 130.17.*.* exceeds Rmax, a new node is created to keep track of packet rates to and from subnets 130.17.0.*, 130.17.1.*, etc. Creating new nodes is called expansion. The reverse, i.e., removing nodes or entire subtrees, is called contraction. Contraction is done when the packet rate from and to a given IP address prefix drop below a certain threshold, or when memory is running out, possibly due to a memory exhaustion attack against MULTOPS itself.
Expansion and contraction enable MULTOPS to exploit the hierarchical structure of the IP address space and the fact that a bandwidth attack is usually directed at (or coming from) a limited set of IP addresses—with a common prefix—only. MULTOPS detects the attack on a high level in the tree (where prefixes are short) and expands toward the largest possible common prefix of the victim’s IP address(es), potentially establishing single IP address(es) that are under attack.
3.5 SOLUTION REQUIREMENTS:

The solution to DDoS can be numerous and various approaches could be used. But what is more important is that a more general and a generic approach will be able to defend it more successfully.

To design a successful defense, we need to first understand the major characteristics of a DDoS attack. The major characteristics of a DDoS attack are as follows:

1. The sheer volume of the packets flowing upwards to the server is indicative of the attack.

2. The packets can be genuine or otherwise but the more important point is that they come from sources located on different interfaces of the router on the upstream.

3. The source IP address of these packets can be one of the invalid addresses as follows:

   0.0.0.0/8    - Historical Broadcast
   10.0.0.0/8   - RFC 1918 Private Network
   127.0.0.0/8  - Loopback
   169.254.0.0/16 - Link Local Networks
   172.16.0.0/12 - RFC 1918 Private Network
   192.0.2.0/24  - TEST-NET
   192.168.0.0/16 - RFC 1918 Private Network
   224.0.0.0/4   - Class D Multicast
   240.0.0.0/5   - Class E Reserved
   248.0.0.0/5   - Unallocated
   255.255.255.255/32 - Broadcast
Defense Against Distributed Denial of Service Attacks

4. Normally there is a prescribed load that any server can bear and hence it automatically puts a limit on the forward and reverse rates of traffic. This is exploited by the attacker. So it becomes imperative to keep the traffic within this threshold.

After analyzing the above characteristics of DDoS attack, we can outline a few solution requirements which could be then fine-tuned to suit particular servers depending on the capabilities of individual servers to handle the traffic.

Outline of the solution:
1. The solution will be implemented on a router level.

2. The router should be provided with additional capabilities like: maintaining a data structure of traffic on the basis of source destination pair, analyzing the traffic rates and hence block the suspected packets but at the same time taking care not to block the genuine users and hence not aid in the denial of service.

3. The processing speed of the router per packet should not be hampered because of the overhead imposed by these additional capabilities.
4. COMPONENT-LEVEL DESIGN
The Software Components designed and implemented are concentrated in 2 areas with respect to the Overall Architecture: The Router and The Graphical Component.

Taking advantage of the modular approach to router architecture provided by Click, we construct a simple IP router with the following characteristics:
1. Packet Generation and Address Modification is possible.
2. Rate selection for packets to or from a particular destination or source.
3. Internal Conversion of packet to Click format allowing greater functionality.
4. Operation in Kernel-mode allows it use TCP/IP stack and ports directly instead of making raw sockets.
4.1 MULTOPS COMPONENT:

The block accepts packets from all sources through the queue and annotates them with the respective forward or reverse rate depending upon the particular source and destination IP address. For every octet of IP address that crosses the threshold value, the tree is unfolded and separate counters are maintained for individual values of the octet to keep track of the rate. When the rate decreases, the tree is folded.

The major functions of the component are:

4.1.1 Push Function

Accepting Input from Push sources and delivering output to Push Destinations.

Figure 4.1 Push Function
4.1.2 Pull Function

Accepting Input from Pull Sources and delivering output to Pull Destinations.

Figure 4.2 Pull Function.
4.1.3 Update Function:

This is responsible for maintaining the Tree data structure of Multops which contains information about the data rates relating to each octet or individual IP addresses. This is implemented through the use of ‘Stats’ and ‘counter’ data structure.

Also it maintains a linear linked list which enables it to keep track of nodes for deletion.

Figure. 4.3 Update Function
4.1.4 Make_Counter Function:

Called by update function to check for memory space and allocate memory for counter data structure. Takes the rate from parent counter structure, previous octet.

Figure 4.4 Make_Counter Function
4.1.5 Fold Function:

performs the actual folding of the tree on the basis of the threshold value.

Figure 4.5. Fold Function.
4.2 DIFFERENTIATOR COMPONENT:
Differentiator element is to be placed downstream of the MULTOPS element. It inspects the forward and reverse rate annotation of the packet as set by the MULTOPS element. These rates it multiplies by the forward and reverse weights respectively. Then depending on the degree of asymmetry between the forward and reverse rates, the packets can be selectively discarded. The forward and reverse weights allow the configuration to control the permissible degree of asymmetry between the two rates.

![Differentiator Diagram](image)

Figure 4.6 Differentiator Component
4.3 ROUTER LOGIC

FromDevice(eth0)

Classifier(…)
ARP ARP IP Queries Response

ARP Responder

To ARP Querier

FromDevice(eth1)

Classifier(…)
ARP ARP IP Queries Response

ARP Responder

To ARP Querier

Strip(14)

CheckIPHeader(…)

MULTOPS

Differentiator

GetIPAddress(16)

LinuxIPRouteLookup(…)

DropBroadcasts

PaintTee(0)

DropBroadcasts

PaintTee(1)
Defense Against Distributed Denial of Service Attacks

- IPGWOptions(…)
- FixIPSource
- Decrement TTL
- IPFragmenter(for Ethernet)
- ARP Querier (for eth 0)
- Queue for eth0
- ToDevice(eth0)

- IPGWOptions(…)
- FixIPSource
- Decrement TTL
- IPFragmenter(for Ethernet)
- ARP Querier (for eth 1)
- Queue for eth0
- ToDevice(eth0)
5. SYSTEM ARCHITECTURE
Defense Against Distributed Denial of Service Attacks
5.2 SYSTEM SPECIFICATION

SYSTEMS USED FOR IMPLEMENTATION:

1. Router Machine:
   Processor : Intel Pentium Processor IV 1.7 GHz
   Onboard RAM : 128MB RD RAM
   Network Adapter : RealTek RTL 8139 (A) PCI Fast Ethernet Adapters (2 nos)
   Operating System : Red Hat Linux 7.1 Server Edition
       Kernel version : 2.4.9
       gcc compiler version : 2.96
   Click Version : 1.4 pre 1

2. LAB Terminals (clients):
   Processor : Intel Pentium II 400 MHz
   Onboard RAM : 128 MB SD RAM
   Network Adapter : RealTek RTL 8139 (A) PCI Fast Ethernet Adapter
   Operating System : Dual OS:
       2. RedHat Linux 7.1 Server Edition
           Kernel version : 2.4.2-1
           gcc compiler version : 2.96
   Click Version : 1.4 pre 1

3. Switch:

   INTEX 8 port n-way Fast Ethernet Switch
6. IMPLEMENTATION
(SOURCE CODES)
6.1 DIFFERENTIATOR.hh:

//HEADER FILE FOR THE PATCHED ELEMENT: DIFFERENTIATOR
//DIFFERENTIATOR IS A DERIVED CLASS FROM THE PARENT CLASS:
//ELEMENT
//IT INHERITS PUBLICLY SEVEN FUNCTIONS FROM ELEMENT
//IT HAS THREE PRIVATE DATA MEMBERS:
//1. _FWD_WEIGHT : TO INPUT THE EXPECTED FORWARD WEIGHT
//2. _REV_WEIGHT : TO INPUT THE EXPECTED REVERSE WEIGHT
//3. _THRESH : TO INPUT THE THRESHOLD FOR SOURCE-DESTINATION PAIR

#ifndef CLICK_DIFFERENTIATOR_HH
#define CLICK_DIFFERENTIATOR_HH
#include <click/element.hh>

CLICK_DECLS

class Differentiator : public Element
{
    private:

        int _fwd_weight;
        int _rev_weight;
        int _thresh;

    public:
        Differentiator();
        void notify_ninputs(int);
        const char *class_name() const
            {return"Differentiator";}
        const char *processing() const
            {return AGNOSTIC;}
        Differentiator *clone() const
            {return new Differentiator;}
        int configure(Vector<String> &conf, ErrorHandler *);
        void push(int port, Packet *);
        Packet *pull(int port);
}

CLICK_ENDDECLS
#endif

6.2 DIFFERENTIATOR . CC:

//A NEW ELEMENT: DIFFERENTIATOR COMPILED AND PATCHED TO CLICK BY
//US.
//DIFFERENTIATOR IS AN ELEMENT HAVING ONE INPUT PORT AND TWO
//OUTPUT PORTS.
//IT IS TO BE PLACED DOWNSTREAM TO THE MULTOPS ELEMENT
//IT DIFFERENTIATES THE INCOMING PACKETS ON THE BASIS OF FORWARD
//AND REVERSE RATES AND A PREDEFINED THRESHOLD.

#include <click/config.h>
#include "differentiator.hh"
#include <click/error.hh>
#include <click/confparse.hh>
#include <click/packet_anno.hh>

CLICK_DECLS

Differentiator :: Differentiator()
{
    _fwd_weight = 0;
    _rev_weight = 0;
}

void
Differentiator::notify_ninputs(int n)
{
    set_ninputs(n = 1);
    set_noutputs(n = 2);
}

int Differentiator :: configure(Vector<String> &conf,
ErrorHandler *errh)
{
    return cp_va_parse(conf, this, errh, cpInteger, "forward weight",
&_fwd_weight, cpInteger, "reverse weight", &_rev_weight,
&Integer, "threshold", &_thresh, 0);
}

void Differentiator :: push(int, Packet *p)
{
    int fwd = p -> user_anno_i(1);
    if (fwd < 1) fwd = 1;
    int rev = p -> user_anno_i(2);
    if (rev < 1) rev = 1;

    if ( (fwd > _thresh || rev > _thresh) && _fwd_weight * fwd
        > _rev_weight * rev)
    {
        output(1).push(p);
    }
    else
Packet * Differentiator :: pull(int)
{
    Packet *p = input(0).pull();
    return p;
}

CLICK_ENDDECLS
ELEMENT_REQUIRES(local)
EXPORT_ELEMENT(Differentiator)
Defense Against Distributed Denial of Service Attacks

6.3 ICMP.CLICK:

//Sample packet generator to be sent to target machine

InfiniteSource(DATA  \<4c 00 10 21  4d 75 4c 00  10 32 43 f9  08 00 45 00
00 54 00 00  40 00 40 01  b9 55 c0 a8  00 02 c0 a8  00 01 08 00  69 82 58 05
00 00 de 25  69 40 f9 0e  0b 00 08 09  0a 0b 0c 0d  0e 0f 10 11  12 13 14 15
16 17 18 19  1a 1b 1c 1d  1e 1f 20 21  22 23 24 25  26 27 28 29  2a 2b 2c 2d
2e 2f 30 31  32 33 34 35  36 37>, LIMIT 5, BURST 1, STOP true)

//Infinite source is an element that generates packet with the
//specified data
//The above packet is a duplicate copy of a genuine ping request
//It has destination IP as the Target machine’s IP address
//The source IP is the IP address of the attacker computer.

-> Print(ATT,100)
-> ToDevice(eth0);
6.4 ATTACK.C:

//PROGRAM TO EXECUTE THE ATTACK BY SENDING PACKETS CONTINUOUSLY
//TO THE TARGET HOST
//IT CALLS THE icmp.click PROGRAM TO GENERATE PING REQUESTS.

#include<stdio.h>
#include<stdlib.h>

int main()
{
    int i = 1;
    char ch;

    printf("Start Attack. Y/N : ");
    ch=getchar();
    if (ch==('y'))
    {
        system("./click-1.4pre1/userlevel/click /icmp");
    }
    else
    {
        printf("\n DID NOT ATTACK\n");
    }
    return(0);
}
6.5 ROUTER SCRIPT:

//ELEMENT DEFINITION

clf1,clf0::Classifier(12/0806 20/0001,12/0806 20/0002,12/0800,-);
//Classifies incoming packets according to category : ARP Queries, ARP Response
// and IP.

arpq1::ARPQuerier(192.168.0.1,4C:00:10:21:4D:75);
//Queries for unknown MAC and sets the Source as this IP 192.168.0.1

arpq0::ARPQuerier(172.16.50.113,4C:00:10:07:18:8A);
// Queries for unknown MAC and sets the Source as this IP 172.16.50.113

q0,q1::Queue;
//Queues for Output to Respective Interfaces

arpr0::ARPResponder(172.16.50.113/255.255.0.0 4C:00:10:07:18:8A,192.168.0.2/
255.255.255.0 4C:00:10:32:43:F9); //Responds to queries for the specified IP Addresses coming on this interface (eth0)

arpr1::ARPResponder(192.168.0.1/255.255.255.0 4C:00:10:21:4D:75,192.168.0.2/
255.255.255.0 4C:00:10:32:43:F9); //Responds to queries for the specified IP Addresses coming on this interface (eth1)

strp::Strip(14);
//Strips off the ethernet header.

chk::CheckIPHeader(0.0.0.0/8 10.0.0.0/8 127.0.0.0/8 169.254.0.0/16 172.16.0.0/12
192.0.2.0/24 192.168.0.0/16 224.0.0.0/4 240.0.0.0/5 248.0.0.0/5 255.255.255.255/32);
//Checks for the specified addresses in the header and discards all the packets with
//these IPs as their Source IPs

getip::GetIPAddress(16);
//Used to Get the Destination IP of the Packet and further helps in table look up.

multops::IPRateMonitor(PACKETS,0.5,1,600);
//MULTOPS Data Structure that maintains the forward and reverse traffic ratios.

diff::Differentiator(1,1,2);
//DIFFERENTIATOR which Classifies the packets as being within limits of
//Threshold or being painted as abnormal packets showing characteristics of
//probable attack.
Defense Against Distributed Denial of Service Attacks

lproute::LinuxIPLookup(eth0,eth1);
//Does the Routing Table Look up

ipoptions0::IPGWOptions(172.16.50.113);
ipoptions1::IPGWOptions(192.168.0.1);
//Processes the IP Options in the Header.

dt0,dt1::DecIPTTL;
//Dekrements the Time To Live field for each incoming packet.

ipf0,ipf1::IPFragmenter(1500);
//Fragments packets if needed

//PROCESSING STARTS HERE

//Input Module
//Accept packets from the interfaces

FromDevice(eth1)->clf1;
FromDevice(eth0)->clf0;

//Classifies into ARP, IP and Other categories.

clf0[0]->arpr0->q0;
clf0[1]->[1]arpq0;

clf1[0] ->arpr1->q1;
clf1[1]->[1]arpq1;

//Routing Process

//IP Packets from the classifier are painted for the incoming interface
//Strip the Ethernet header
clf1[2]->Paint(1)->strp;
clf0[2]->Paint(0)->strp;
//Rate Monitoring Process
strp
  ->chk  //Check Bad source IP
  ->multops  //Register in the data structure
  ->diff;  //Differentiate based on rate

//Packets displaying attack characteristics
diff[1]
  ->Print(PACKET_REJECTED,64)
  ->GetIPAddress(16)
  ->Print(IP_address)
  ->ToDump(packet.dat , ENCAP IP)  //Dump the Packets in a file
       //for further analysis and tracking
  ->Discard;  //Discards the packets suspected of attack

//Routing of Genuine Packets
diff[0]
  ->getip
  ->lproute;

lproute[0]
  ->DropBroadcasts  //Drops Broadcast packets
  ->pt1::PaintTee(0);

lproute[1]
  ->DropBroadcasts
  ->pt2::PaintTee(1);

pt1[0]
  ->ipoptions0[0]
  ->FixIPSrc(172.16.50.113)
  ->dt0[0]
  ->ipf0[0]
  ->[0]arpq0;

pt2[0]
  ->ipoptions1[0]
  ->FixIPSrc(192.168.0.1)
  ->dt1[0]
  ->ipf1[0]
  ->[0]arpq1;
// ERROR HANDLING MODULE

// Network Unreachable error
lproute[2]
    ->ICMPError(172.16.50.113, 3, 0)
    ->lproute;

// Redirect for Host
pt1[1]
    ->ICMPError(172.16.50.113,5,1)
    ->lproute;
pt2[1]
    ->ICMPError(192.168.0.1,5,1)
    ->lproute;

// IP header bad (catchall error)
ipoptions0[1]
    ->ICMPError(172.16.50.113,12,0)
    ->lproute;
ipoptions1[1]
    ->ICMPError(192.168.0.1,12,0)
    ->lproute;

// TTL equals 0 during transit
dt0[1]
    ->ICMPError(172.16.50.113, 11, 0)
    ->lproute;
dt1[1]
    ->ICMPError(192.168.0.1, 11, 0)
    ->lproute;

// Fragmentation needed but no frag. bit set
ipf0[1]
    ->ICMPError(172.16.50.113, 3, 4)
    ->lproute;

ipf1[1]
    ->ICMPError(192.168.0.1, 3, 4)
    ->lproute;
//OUTPUT MODULE
arpq0->q0->ToDevice(eth0);
apql->q1->ToDevice(eth1);
clf1[3]->Discard;
clf0[3]->Discard;
6.6 FOR VIEWING LOG

//PROGRAM TO CREATE A LOG OF DISCARDED PACKETS DUE TO //ABNORMAL TRAFFIC RATE

#include <stdio.h>
#include <stdlib.h>

//Two files are being used:
//Packet.dat is the file in which the router script dumps the
//discarded packets.

//temp.dat where we extract the source and destination IP of
//each discarded packet and later use this file to get a count
//to get total packets for each source destination pair.

FILE * f1;
FILE * f2;

struct packet
{
    char ip[23];
    long count;
    struct packet *prev,*next;
};

int main()
{
    struct packet *first,*trav,*last, *newrec, *old;
    int count=0,i,flag=1,fl1=0,fl2=0;
    char c, arr[23];

    system("clear");
    printf("\n TABLE SHOWING IP PACKETS THAT WERE BLOCKED AND THEIR
COUNT\n");
    f1=fopen("/packet.dat","r");
    f2=fopen("/temp.dat","w");
while((c=getc(f1))!='\n');

//This part of the code extracts the source and destination IP from each packet
while((c=getc(f1))!=EOF)
{
    count ++;
    if(count==37) fprintf(f2, "\n");
    if((count>28)&&(count<37))
    {
        fprintf(f2, "%x", ((c&0xf0)/16));
        fprintf(f2, "%1x", (c&0x0f));
    }
    if(c=='5') {fl1=1;}
    if((c=='6')&&(fl1==1)) {fl2=1; }
    if((c=='7')&&(fl1==1)&&(fl2==1)) {count=0;fl1=fl2=0;}
        //if(count==100) count=0;
}
fclose(f1);
fclose(f2);

//We use a linked list to maintain a record of all source-destination pairs and their respective count of packets
f2=fopen("/temp.dat","r");
first= (struct packet*)malloc(sizeof(struct packet));
first->ip[0] = '\0';
first->prev=NULL;
first->next=NULL;
first->count=0;
count=0;
old=first;
trav=first;

while ((c=getc(f2))!= EOF)
{
    if((count<23)&&(count>=0))
    {
        arr[count]=c;
    }
}
Defense Against Distributed Denial of Service Attacks

if(count==23) {
    if((first->ip[0])==\'0\')
    {
        for(i=0;i<23;i++)
            first->ip[i] = arr[i];
        first->count=1;
    }
    else
    {
        while(trav != NULL)
        {
            flag=1;
            for(i=0;i<23;i++)
                if(trav->ip[i] != arr[i]) flag = 0;
                //0 is for different
            if(flag==1) (trav->count)++;  
                        //1 is for same
            trav=trav->next;
        }
        if(flag == 0)
        {
            newrec=(struct packet*)malloc(sizeof(struct packet));
            for(i=0;i<23;i++)
                newrec->ip[i] = arr[i];
            newrec->next=NULL;
            newrec->count=1;
            old->next=newrec;
            newrec->prev=old;
            old=newrec;
        }
    }
    count++;
    if(count==25) {count=0; flag=1; trav=first;}
}
fclose(f2);
printf("\n\n");
trav=first->next;
// This part prints the source destination and the count.
while(trav != NULL)
{
    printf("From Source : ");
    for(i=0;i<12;i++)
        printf("%c",trav->ip[i]);
    printf(" To Destination : ");
    for(i=12;i<23;i++)
        printf("%c",trav->ip[i]);
    printf(" count is : %ld 
",(trav->count));
    trav=trav->next;
}

printf("\n\n");

return(0);
7. APPENDIX
Defense Against Distributed Denial of Service Attacks

**IMPLEMENTATION PRE-REQUISITES:**

1. Upgrading Default kernel 2.4.2-2 to 2.4.9
2. Installing Click 1.4 pre-1 in Kernel mode by patching Click 1.4 pre-1 kernel patch into kernel 2.4.9
3. Compiling and patching the new element “differentiator” into Click.

**7.1 Upgrading Default kernel 2.4.2-2 to 2.4.9**

**Installing the kernel:**

1. Unpack the tar.gz kernel file
   
   gzip -cd linux-2.4.9.tar.gz | tar xvf -

2. Do a “make menuconfig” to configure the basic kernel
   
   4. Finally, do a “make dep
   5. .” to set up all the dependencies correctly.

**Compiling the kernel:**

1. Do a “make bzImage” to create a compressed kernel image.

2. If any of the parts of the kernel have been configured as ‘modules’, then “make modules” should be followed by “make modules_install”.
3. In order to boot the new kernel, there’s a need to copy the kernel image (found in `/linux/arch/i386/boot/bzImage` after compilation) to the place where the regular bootable kernel is found.

4. If Linux is booted from the hard drive, the kernel image as specified in the file `/etc/lilo.conf`. The kernel image file is usually `/vmlinuz`, `/boot/vmlinuz`, `/bzImage` or `/boot/bzImage`. To use the new kernel, save a copy of the old image and copy the new image over the old one. RERUN LILO to update the loading map.

5. After reinstalling LILO shutdown the system and reboot.

7.2 Installing Click 1.4 pre-1 in Kernel mode by patching Click 1.4 pre-1 kernel patch into kernel 2.4.9

**Configuring:**

1. Run `./configure` [—prefix=PREFIX]

2. After running `./configure`, the ‘make install’ command installs the user-level executable ‘click’, the kernel module ‘click.o’, and configuration optimizers.

**Userlevel**

1. Build the user-level Click program in the ‘userlevel’ directory:

   ```
   cd CLICKDIR/userlevel
   gmake
   ```

   This will eventually create an executable named ‘click’.
Defense Against Distributed Denial of Service Attacks

**LINUX MODULE:**

First we install, patch, compile, and boot a compatible Linux kernel with click. We use Kernel 2.4.9

1. Archive a complete, working Linux kernel and any corresponding modules.

3. Install the Click Linux kernel patch:
   
   ```
   cd /usr/src/linux
   patch -p0 -b < CLICKDIR/etc/linux-2.4.9-patch
   ```

5. Configure the new kernel by ‘make menuconfig’. Use a minimal set of options.

6. Compile and install the kernel:

   ```
   make dep
   make bzImage
   make install
   make modules
   make modules_install
   ```

7. Edit `/etc/lilo.conf` to tell it about the new kernel, then run `/sbin/lilo`.

8. Reboot your machine with the new kernel.
9. Rerun './configure’ to tell the system about your new kernel:

   rm -f config.cache ; ./configure [OPTIONS]

10. Then build and install the click module, its companion proclikefs
    module, and the language tools:

    gmake install

7.3 Compiling and patching the new element “differentiator” into Click.

Steps for including new element Differentiator in Click:
1. Writing the new element class:
   New element class – Differentiator is written as two C++ source files, differentiator.cc and
differentiator.hh
   The .cc file exports the element class with EXPORT_ELEMENT.
   EXPORT_ELEMENT(Differentiator)

   EXPORT_ELEMENT takes a single argument, the name of the C++ class corresponding to
differentiator element.

2. Put differentiator element (both .cc and .hh files) in an ‘elements/’ directory.
3. run ‘make elemlist’.

   ‘make elemlist’ checks the source files in the ‘elements/’ subdirectories for
   EXPORT_ELEMENT directives, and compiles a list of elements that Click should compile.
   After running ‘make elemlist’, check the ‘userlevel/elements.conf’ and ‘linuxmodule/
elements.conf’ files to see if differentiator.cc file made it into this list.

4. run ‘make install’