**TRACE BACK OF DDOS ATTACKS USING ENTROPY VARIATIONS**

By

TAMARA SAAD MOHAMED

Reg No : 3551010031

A Project Report

Submitted in partial fulfillment of the requirements

for the award of the degree of

**M.Sc in Information Technology**



**SCHOOL OF COMPUTER SCIENCE**

**FACULTY OF SCIENCE & HUMANITIES**

**SRM UNIVERSITY**

**KATTANKULATHUR – 603 203**

**MAY 2012**

**TRACE BACK OF DDOS ATTACKS USING**

**ENTROPY VARIATIONS**

1. **INTRODUCTION**

It is an extraordinary challenge to traceback the source of Distributed Denial-of-Service (DDoS) attacks in the Internet. In DDoS attacks, attackers generate a huge amount of requests to victims through compromised computers (zombies), with the aim of denying normal service or degrading of the quality of services. It has been a major threat to the Internet since year 2000, and a recent survey on the largest 70 Internet operators in the world demonstrated that DDoS attacks are increasing dramatically, and individual attacks are more strong and sophisticated.

Furthermore, the survey also found that the peak of 40 gigabit DDoS attacks nearly doubled in 2008 compared with the previous year. The key reason behind this phenomena is that the network security community does not have effective and efficient traceback methods to locate

attackers as it is easy for attackers to disguise themselves by taking advantages of the vulnerabilities of the World Wide Web, such as the dynamic, stateless, and anonymous nature

of the Internet. IP traceback means the capability of identifying the actual source of any packet sent across the Internet. Because of the vulnerability of the original design of the Internet, we may not be able to find the actual hackers at present. In fact, IP traceback schemes are considered successful if they can identify the zombies from which the DDoS attack packets entered the Internet. Research on DDoS detection, mitigation, and filtering has been conducted pervasively. However, the efforts on IP traceback are limited. A number of IP traceback approaches have been suggested to identify attackers and there are two major methods for IP traceback, the probabilistic packet marking (PPM) and the deterministic packet marking (DPM). Both of these strategies require routers to inject marks into individual packets. Moreover, the PPM strategy can only operate in a local range of the Internet (ISP network), where the defender has the authority to manage. However, this kind of ISP networks is generally quite small, and we cannot traceback to the attack sources located out of the ISP network. The DPM strategy requires all the Internet routers to be updated for packet marking. However, with only 25 spare bits available in as IP packet, the scalability of DPM is a huge problem. Moreover, the DPM mechanism poses an extraordinary challenge on storage for packet logging for routers.Therefore, it is infeasible in practice at present. Further, both PPM and DPM are vulnerable to hacking, which is referred to as packet pollution. IP traceback methods should be independent of packet pollution and various attack patterns. In our previous work, on DDoS attack detection, we compared the packet number distributions of packet flows, which are out of the control of attackers once the attack is launched, and we found that the similarity of attack flows is much higher than the similarity among legitimate flows, e.g., flash crowds. Entropy rate, the entropy growth rate as the length of a stochastic sequence increases, was employed to find the similarity between two flows on the entropy growth pattern, and relative entropy, an abstract distance between two probabilistic mass distributions, was taken to measure the instant difference between two flows. In this paper, we propose a novel mechanism for IP traceback using information theoretical parameters, and there is no packet marking in the proposed strategy; we, therefore, can avoid the inherited shortcomings of the packet marking mechanisms. We categorize packets that are passing through a router into flows, which are defined by the upstream router where a packet came from, and the destination address of the packet. During nonattack periods, routers are required to observe and record entropy variations of local flows. In this paper, we use flow entropy variation or entropy variation interchangeably. Once a DDoS

attack has been identified, the victim initiates the following pushback process to identify the locations of zombies: the victim first identifies which of its upstream routers are in the attack tree based on the flow entropy variations it has accumulated, and then submits requests to the related

immediate upstream routers. The upstream routers identify where the attack flows came from based on their local entropy variations that they have monitored. Once the immediate upstream routers have identified the attack flows, they will forward the requests to their immediate upstream routers, respectively, to identify the attacker sources further; this procedure is repeated in a parallel and distributed fashion until it reaches the attack source or the discrimination limit between attack flows and legitimate flows is satisfied. Our analysis, experiments, and simulations demonstrate that the proposed traceback mechanism is effective and efficient compared with the existing methods. In particular, it possesses the following advantages,

* The proposed strategy is fundamentally different from the existing PPM or DPM traceback mechanisms, and it outperforms the available PPM and DPM methods. Because of this essential change, the proposed strategy overcomes the inherited drawbacks of packet marking methods, such as limited scalability, huge demands on storage space, and vulnerability to packet pollutions.
* The implementation of the proposed method brings no modifications on current routing software. Both PPM and DPM require update on the existing routing software, which is extremely hard to achieve on the Internet. On the other hand, our proposed method can work independently as an additional module on routers for monitoring and recording flow information, and communicating with its upstream and downstream routers when the pushback procedure is carried out.
* The proposed method will be effective for future packet flooding DDoS attacks because it is independent of traffic patterns. Some previous works depend heavily on traffic patterns to conduct their traceback. For example, they expected that traffic patterns obey Poisson distribution or Normal distribution. However, traffic patterns have no impact on the proposed scheme; therefore, we can deal with any complicated attack patterns, even legitimate traffic pattern mimicking attacks.
* The proposed method can archive real-time traceback to attackers. Once the short-term flow information is in place at routers, and the victim notices that it is under attack, it will start the traceback procedure. The workload of traceback is distributed, and the overall traceback time mainly depends on the network delays between the victim and the attackers.

**ABSTRACT 2. ABSTRACT**

Distributed Denial-of-Service (DDoS) attacks are a critical threat to the Internet. However, the memoryless feature of the Internet routing mechanisms makes it extremely hard to Traceback to the source of these attacks. As a result, there is no effective and efficient method to deal with this issue so far. In this paper, we propose a novel Traceback method for DDoS attacks that is based on entropy variations between normal and DDoS attack traffic, which is fundamentally different from commonly used packet marking techniques. In comparison to the existing DDoS Traceback methods, the proposed strategy possesses a number of advantages—it is memory nonintensive, efficiently scalable, robust against packet pollution, and independent of attack traffic patterns. The results of extensive experimental and simulation studies are presented to demonstrate the effectiveness and efficiency of the proposed method our experiments show that accurate Traceback is possible within 20 seconds (approximately) in a large-scale attack network with thousands of zombies.

**ORGNAIZATION**

**PROFILE**

**3. ORGANIZATION PROFILE**

At Blue Chip Technologies, We go beyond providing software solutions. We work with our client’s technologies and business changes that shape their competitive advantages.

Founded in 2000, Blue Chip Technologies (P) Ltd. is a software and service provider that helps organizations deploy, manage, and support their business-critical software more effectively. Utilizing a combination of proprietary software, services and specialized expertise, Blue Chip Technologies (P) Ltd. helps mid-to-large enterprises, software companies and IT service providers improve consistency, speed, and transparency with service delivery at lower costs. Blue Chip Technologies (P) Ltd. helps companies avoid many of the delays, costs and risks associated with the distribution and support of software on desktops, servers and remote devices. Our automated solutions include rapid, touch-free deployments, ongoing software upgrades, fixes and security patches, technology asset inventory and tracking, software license optimization, application self-healing and policy management. At Blue Chip Technologies, we go beyond providing software solutions. We work with our clients’ technologies and business processes that shape their competitive advantages.

**ABOUT THE PEOPLE**

As a team we have the prowess to have a clear vision and realize it too. As a statistical evaluation, the team has more than 40,000 hours of expertise in providing real-time solutions in the fields of Embedded Systems, Control systems, Micro-Controllers, c Based Interfacing, Programmable Logic Controller, VLSI Design And Implementation, Networking With C, ++, java, client Server Technologies in Java,(J2EE\J2ME\J2SE\EJB),VB & VC++, Oracle and operating system concepts with LINUX.

**OUR VISION**

“Dreaming a vision is possible and realizing it is our goal”.

**OUR MISSION**

We have achieved this by creating and perfecting processes that are in par with the global standards and we deliver high quality, high value services, reliable and cost effective IT products to clients around the world.

**CLIENTS**

* Aray InfoTech
* Inquirre consultancy (U.S.A)
* K square consultancy pvt Ltd (U.S.A)
* Opal solutions
* Texlab Solutions
* Vertex Business Machines
* JM InfoTech

**SYSTEM ANALYSIS**

**4. SYSTEM ANALYSIS**

* 1. **. EXISTING SYSTEM**

It is an extraordinary challenge to Traceback the source of Distributed Denial-of-Service (DDoS) attacks in the Internet. A number of IP Traceback approaches have been suggested to identify attackers and there are two major methods for IP Traceback, the probabilistic packet marking (PPM) and the deterministic packet marking (DPM). Moreover, the PPM strategy can only operate in a local range of the Internet (ISP network) where the defender has the authority to manage. However, this kind of ISP networks is generally quite small, and we cannot Traceback to the attack sources located out of the ISP network. The DPM strategy requires all the Internet routers to be updated for packet marking. However, with only 25 spare bits available in as IP packet, the scalability of DPM is a huge problem. Moreover, the DPM mechanism poses an extraordinary challenge on storage for packet logging for routers. Further, both PPM and DPM are vulnerable to hacking, which is referred to as packet pollution.

**DIS-ADVANTAGE**

* There is no effective and efficient method to deal with this issue so far.
* It is infeasible in practice at present.
* Work depends heavily on traffic patterns to conduct their Traceback.

**4.2. PROPOSED SYSTEM**

We propose a novel Traceback method for DDoS attacks that is based on entropy variations between normal and DDoS attack traffic, which is fundamentally different from commonly used packet marking techniques. IP Traceback means the capability of identifying the actual source of any packet sent across the Internet. In fact IP Traceback schemes are considered successful if they can identify the zombies from which the DDoS attack packets entered the Internet. IP Traceback methods should be independent from packet pollution and various attack patterns. Entropy rate, the entropy growth rate as the length of a stochastic sequence increases, was employed to find the similarity between two flows on the entropy growth pattern, and relative entropy, an abstract distance between two probabilistic mass distributions, was taken to measure the instant difference between two flows. Our proposed method can work independently as an additional module on routers for monitoring and recording flow information, and communicating with its upstream and downstream routers when the pushback procedure is carried out. The proposed method can archive real time traceback to attackers. Once the short term flow information is in place at routers, and the victim notices that it is under attack, it will start the Traceback procedure. The workload of traceback is distributed, and the overall Traceback time mainly depends on network delays between the victim and the attackers.

**ADVANTAGE**

* In comparison to existing DDoS Traceback methods, the proposed strategy possesses a number of advantages - it is memory non-intensive, efficiently scalable, robust against packet pollution and independent of attack traffic patterns.
* The results of extensive experimental and simulation studies are presented to demonstrate the effectiveness and efficiency of the proposed method.
* Our experiments show that accurate Traceback is possible within 20 seconds (approx.) in a large scale attack network with thousands of zombies.
* The proposed strategy overcomes the inherited drawbacks of packet marking methods, such as limited scalability, huge demands on storage space and vulnerability to packet pollutions
* The proposed method will be effective for future packet flooding DDoS attacks because it is independent of traffic patterns.

**SYSTEM SPECIFICATION**

**5. SYSTEM SPECIFICATION**

* 1. **HARDWARE REQUIREMENTS**

Hard disk : 80 GB

RAM : 512MB or more

Processor Speed : 3.00GHz

Processor : Pentium IV Processor or more

* 1. **SOFTWARE REQUIREMENTS**

Operating System : Win 2000 and above

Front-End : Java Swings

Backend : SQL Server 2000

Implementation Concept : Socket in java

* 1. **5.3. SOFTWARE DESCRIPTION**

**ABOUT JAVA**

**Java** is a programming language originally developed by Sun Microsystems and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of computer architecture.

One characteristic of Java is portability, which means that computer programs written in the Java language must run similarly on any supported hardware/operating-system platform. One should be able to write a program once, compile it once, and run it anywhere.

This is achieved by compiling the Java language code, not to machine code but to Java bytecode – instructions analogous to machine code but intended to be interpreted by a virtual machine (VM) written specifically for the host hardware. End-users commonly use a JRE installed on their own machine, or in a Web browser.

Standardized libraries provide a generic way to access host specific features such as graphics, threading and networking. In some JVM versions, bytecode can be compiled to native code, either before or during program execution, resulting in faster execution.

A major benefit of using bytecode is porting. However, the overhead of interpretation means that interpreted programs almost always run more slowly than programs compiled to native executables would, and Java suffered a reputation for poor performance. This gap has been narrowed by a number of optimization techniques introduced in the more recent JVM implementations.

One such technique, known as (just-in-time compilation) JIT, translates Java bytecode into native code the first time that code is executed, then caches it. This result in a program that starts and executes faster than pure interpreted code can, at the cost of introducing occasional compilation overhead during execution. More sophisticated VMs also use dynamic recompilation, in which the VM analyzes the behavior of the running program and selectively recompiles and optimizes parts of the program. Dynamic recompilation can achieve optimizations superior to static compilation because the dynamic compiler can base optimizations on knowledge about the runtime environment and the set of loaded classes, and can identify *hot spots* - parts of the program, often inner loops, that take up the most execution time. JIT compilation and dynamic recompilation allow Java programs to approach the speed of native code without losing portability.

Another technique, commonly known as *static compilation*, or ahead-of-time (AOT) compilation, is to compile directly into native code like a more traditional compiler. Static Java compilers translate the Java source or bytecode to native object code. This achieves good performance compared to interpretation, at the expense of portability; the output of these compilers can only be run on a single architecture. AOT could give Java something like performance, yet it is still not portable since there are no compiler directives, and all the pointers are indirect with no way to micro manage garbage collection.

Java's performance has improved substantially since the early versions, and performance of JIT compilers relative to native compilers has in some tests been shown to be quite similar. The performance of the compilers does not necessarily indicate the performance of the compiled code; only careful testing can reveal the true performance issues in any system.

One of the unique advantages of the concept of a runtime engine is that errors (exceptions) should not 'crash' the system. Moreover, in runtime engine environments such as Java there exist tools that attach to the runtime engine and every time that an exception of interest occurs they record debugging information that existed in memory at the time the exception was thrown (stack and heap values). These Automated Exception Handling tools provide 'root-cause' information for exceptions in Java programs that run in production, testing or development environments.

**SWINGS:**

Swing is a platform-independent, ***Model-View-Controller*** GUI framework for Java. It follows a single-threaded programming model, and possesses the following traits:

**PLATFORM INDEPENDENCE:**

Swing is platform independent both in terms of its expression (Java) and its implementation (non-native universal rendering of widgets).

**EXTENSIBILITY:**

Swing is a highly partitioned architecture, which allows for the "plugging" of various custom implementations of specified framework interfaces: Users can provide their own custom implementation(s) of these components to override the default implementations. In general, Swing users can extend the framework by extending existing (framework) classes and/or providing alternative implementations of core components.

**COMPONENT-ORIENTED :**

Swing is a component-based framework. The distinction between objects and components is a fairly subtle point: concisely, a component is a well-behaved object with a known/specified characteristic pattern of behaviour. Swing objects asynchronously fire events, have "bound" properties, and respond to a well-known set of commands (specific to the component.) Specifically, Swing components are Java Beans components, compliant with the Java Beans Component Architecture specifications.

**CUSTOMIZABLE:**

Given the programmatic rendering model of the Swing framework, fine control over the details of rendering of a component is possible in Swing. As a general pattern, the visual representation of a Swing component is a composition of a standard set of elements, such as a "border", "inset", decorations, etc. Typically, users will programmatically customize a standard Swing component (such as a JTable) by assigning specific Borders, Colors, Backgrounds, opacities, etc., as the properties of that component. The core component will then use these property (settings) to determine the appropriate renderers to use in painting its various aspects. However, it is also completely possible to create unique GUI controls with highly customized visual representation.

**CONFIGURABLE:**

Swing's heavy reliance on runtime mechanisms and indirect composition patterns allows it to respond at runtime to fundamental changes in its settings. For example, a Swing-based application can change its look and feel at runtime. Further, users can provide their own look and feel implementation, which allows for uniform changes in the look and feel of existing Swing applications without any programmatic change to the application code.

**LIGHTWEIGHT UI :**

Swing's configurability is a result of a choice not to use the native host OS's GUI controls for displaying itself. Swing "paints" its controls programmatically through the use of Java 2D APIs, rather than calling into a native user interface toolkit. Thus, a Swing component does not have a corresponding native OS GUI component, and is free to render itself in any way that is possible with the underlying graphics APIs.

However, at its core every Swing component relies on an AWT container, since (Swing's) JComponent extends (AWT's) Container. This allows Swing to plug into the host OS's GUI management framework, including the crucial device/screen mappings and user interactions, such as key presses or mouse movements. Swing simply "transposes" its own (OS agnostic) semantics over the underlying (OS specific) components. So, for example, every Swing component paints its rendition on the graphic device in response to a call to component.paint(), which is defined in (AWT) Container. But unlike AWT components, which delegated the painting to their OS-native "heavyweight" widget, Swing components are responsible for their own rendering.

This transposition and decoupling is not merely visual, and extends to Swing's management and application of its own OS-independent semantics for events fired within its component containment hierarchies. Generally speaking, the Swing Architecture delegates the task of mapping the various flavors of OS GUI semantics onto a simple, but generalized, pattern to the AWT container. Building on that generalized platform, it establishes its own rich and complex GUI semantics in the form of the JComponent model. A review of the source of Container. java and JComponent.java classes is recommended for further insights into the nature of the interface between Swing's lightweight components and AWT's heavyweight widgets.

**LOOSELY-COUPLED/MVC:**

The Swing library makes heavy use of the Model/View/Controller software design pattern, which conceptually decouples the data being viewed from the user interface controls through which it is viewed. Because of this, most Swing components have associated *models* (which are specified in terms of Java interfaces), and the programmer can use various default implementations or provide their own. The framework provides default implementations of model interfaces for all of its concrete components.

Typically, Swing component model objects are responsible for providing a concise interface defining events fired, and accessible properties for the (conceptual) data model for use by the associated JComponent. Given that the overall MVC pattern is a loosely-coupled collaborative object relationship pattern, the model provides the programmatic means for attaching event listeners to the data model object. Typically, these events are model centric (ex: a "row inserted" event in a table model) and are mapped by the JComponent specialization into a meaningful event for the GUI component.

For example, the JTable has a model called TableModel that describes an interface for how a table would access tabular data. A default implementation of this operates on a two-dimensional array.

The view component of a Swing JComponent is the object used to graphically "represent" the conceptual GUI control. A distinction of Swing, as a GUI framework, is in its reliance on programmatically-rendered GUI controls (as opposed to the use of the native host OS's GUI controls). This distinction is a source of complications when mixing AWT controls, which use native controls, with Swing controls in a GUI.

Finally, in terms of visual composition and management, Swing favors relative layouts (which specify the positional relationships between components) as opposed to absolute layouts (which specify the exact location and size of components). This bias towards "fluid"' visual ordering is due to its origins in the applet operating environment that framed the design and development of the original Java GUI toolkit. (Conceptually, this view of the layout management is quite similar to that which informs the rendering of HTML content in browsers, and addresses the same set of concerns that motivated the former.)

### LOOK AND FEEL:

Swing allows one to specialize the look and feel of widgets, by modifying the default (via runtime parameters), deriving from an existing one, by creating one from scratch, or, beginning with **J2SE 5.0**, by using the skinnable synth Look and Feel (see Synth Look and Feel), which is configured with an XML property file. The look and feel can be changed at runtime, and early demonstrations of Swing frequently provided a way to do this.

### RELATIONSHIP TO AWT:

Since early versions of Java, a portion of the Abstract Window Toolkit (AWT) has provided platform-independent APIs for user interface components. In AWT, each component is rendered and controlled by a native peer component specific to the underlying windowing system.

By contrast, Swing components are often described as *lightweight* because they do not require allocation of native resources in the operating system's windowing toolkit. The AWT components are referred to as *heavyweight components*.

Much of the Swing API is generally a complementary extension of the AWT rather than a direct replacement. In fact, every Swing lightweight interface ultimately exists within an AWT heavyweight component because all of the top-level components in Swing (JApplet, JDialog, JFrame, and JWindow) extend an AWT top-level container. However, the use of both lightweight and heavyweight components within the same window is generally discouraged due to Z-order incompatibilities.

The core rendering functionality used by Swing to draw its lightweight components is provided by Java 2D, another part of JFC.

**NETWORKING BASICS:**

Ken Thompson and Dennis Ritchie developed UNIX in concert with the C language at Bell Telephone Laboratories, Murray Hill, New Jersey, in 1969. In 1978, Bill Joy was leading a project at Cal Berkeley to add many new features to UNIX, such as virtual memory and full-screen display capabilities. By early 1984, just as Bill was leaving to found Sun Microsystems, he shipped 4.2BSD, commonly known as Berkeley UNIX.4.2BSD came with a fast file system, reliable signals, interprocess communication, and, most important, networking. The networking support first found in 4.2 eventually became the de facto standard for the Internet. Berkeley’s implementation of TCP/IP remains the primary standard for communications with the Internet. The socket paradigm for inter process and network communication has also been widely adopted outside of Berkeley.

**SOCKET OVERVIEW:**

A *network socket* is a lot like an electrical socket. Various plugs around the network have a standard way of delivering their payload. Anything that understands the standard protocol can “plug in” to the socket and communicate.

*Internet protocol (IP)* is a low-level routing protocol that breaks data into small packets and sends them to an address across a network, which does not guarantee to deliver said packets to the destination.

*Transmission Control Protocol* (TCP) is a higher-level protocol that manages to reliably transmit data. A third protocol, *User Datagram Protocol (UDP)*, sits next to TCP and can be used directly to support fast, connectionless, unreliable transport of packets.

**CLIENT/SERVER:**

A *server* is anything that has some resource that can be shared. There are *compute servers*, which provide computing power; *print servers*, which manage a collection of printers; *disk servers*, which provide networked disk space; and *web servers*, which store web pages. A *client* is simply any other entity that wants to gain access to a particular server.

In Berkeley sockets, the notion of a socket allows as single computer to serve many different clients at once, as well as serving many different types of information. This feat is managed by the introduction of a *port*, which is a numbered socket on a particular machine. A server process is said to “listen” to a port until a client connects to it. A server is allowed to accept multiple clients connected to the same port number, although each session is unique. To mange multiple client connections, a server process must be multithreaded or have some other means of multiplexing the simultaneous I/O.

**RESERVED SOCKETS:**

Once connected, a higher-level protocol ensues, which is dependent on which port you are using. TCP/IP reserves the lower, 1,024 ports for specific protocols. Port number 21 is for FTP, 23 is for Telnet, 25 is for e-mail, 79 is for finger, 80 is for HTTP, 119 is for Netnews-and the list goes on. It is up to each protocol to determine how a client should interact with the port.

Java and the Net

Java supports TCP/IP both by extending the already established stream I/O interface. Java supports both the TCP and UDP protocol families. TCP is used for reliable stream-based I/O across the network. UDP supports a simpler, hence faster, point-to-point datagram-oriented model.

InetAddress

The InetAddress class is used to encapsulate both the numerical IP address and the domain name for that address. We interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The InetAddress class hides the number inside. As of Java 2, version 1.4, InetAddress can handle both IPv4 and IPv6 addresses.

**FACTORY METHODS:**

The InetAddress class has no visible constructors. To create an InetAddress object, we use one of the available factory methods. *Factory methods* are merely a convention whereby static methods in a class return an instance of that class. This is done in lieu of overloading a constructor with various parameter lists when having unique method names makes the results much clearer.

Three commonly used InetAddress factory methods are shown here.

static InetAddress getLocalHost( ) throws UnknownHostException

static InetAddress getByName(String hostName) throws UnknowsHostException

static InetAddress[ ] getAllByName(String hostName)

throws UnknownHostException

The getLocalHost( ) method simply returns the InetAddress object that represents the local host. The getByName( ) method returns an InetAddress for a host name passed to it. If these methods are unable to resolve the host name, they throw an UnknownHostException.

On the internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The getAllByName ( ) factory method returns an array of InetAddresses that represent all of the addresses that a particular name resolves to. It will also throw an UnknownHostException if it can’t resolve the name to at least one address. Java 2, version 1.4 also includes the factory method getByAddress( ), which takes an IP address and returns an InetAddress object. Either an IPv4 or an IPv6 address can be used.

**INSTANCE METHODS:**

The InetAddress class also has several other methods, which can be used on the objects returned by the methods just discussed. Here are some of the most commonly used.

Boolean equals (Object other)- Returns true if this object has the same Internet address as other.

byte[ ] getAddress( )- Returns a byte array that represents the object’s Internet address in network byte order.

String getHostAddress( )- Returns a string that represents the host address associated with the InetAddress object.

String getHostName( )- Returns a string that represents the host name associated with the InetAddress object.

boolean isMulticastAddress( )- Returns true if this Internet address is a multicast address. Otherwise, it returns false.

String toString( )- Returns a string that lists the host name and the IP address for conveneince.

Internet addresses are looked up in a series of hierarchically cached servers. That means that your local computer might know a particular name-to-IP-address mapping autocratically, such as for itself and nearby servers. For other names, it may ask a local DNS server for IP address information. If that server doesn’t have a particular address, it can go to a remote site and ask for it. This can continue all the way up to the root server, called InterNIC (internic.net).

**TCP/IP CLIENT SOCKETS:**

TCP/IP sockets are used to implement reliable, bidirectional, persistent, point-to-point, stream-based connections between hosts on the Internet. A socket can be used to connect Java’s I/O system to other programs that may reside either on the local machine or on any other machine on the Internet.

There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The ServerSocket class is designed to be a “listener,” which waits for clients to connect before doing anything. The Socket class is designed to connect to server sockets and initiate protocol exchanges.

The creation of a Socket object implicitly establishes a connection between the client and server. There are no methods or constructors that explicitly expose the details of establishing that connection. Here are two constructors used to create client sockets:

Socket(String *hostName*, int *port*) Creates a socket connecting the local host to the named host and port; can throw an UnknownHostException or anIOException.

Socket(InetAddress *ipAddress*, int *port*) Creates a socket using a preexisting InetAddress object and a port; can throw an IOException.

A socket can be examined at any time for the address and port information associated with it, by use of the following methods:

InetAddress getInetAddress( )- Returns the InetAddress associated with the Socket object.

int getPort( ) Returns the remote port to which this Socket object is connected.

int getLocalPort( ) Returns the local port to which this Socket object is connected.

Once the Socket object has been created, it can also be examined to gain access to the input and output streams associated with it. Each of these methods can throw an IOException if the sockets have been invalidated by a loss of connection on the Net.

InputStream getInputStream( )Returns the InputStream associated with the invoking socket.

OutputStream getOutputStream( ) Returns the OutputStream associated with the invoking socket.

**TCP/IP SERVER SOCKETS:**

Java has a different socket class that must be used for creating server applications. The ServerSocket class is used to create servers that listen for either local or remote client programs to connect to them on published ports. ServerSockets are quite different form normal Sockets.

When we create a ServerSocket, it will register itself with the system as having an interest in client connections. The constructors for ServerSocket reflect the port number that we wish to accept connection on and, optionally, how long we want the queue for said port to be. The queue length tells the system how many client connection it can leave pending before it should simply refuse connections. The default is 50. The constructors might throw an IOException under adverse conditions. Here are the constructors:

ServerSocket(int port) Creates server socket on the specified port with a queue length of 50.

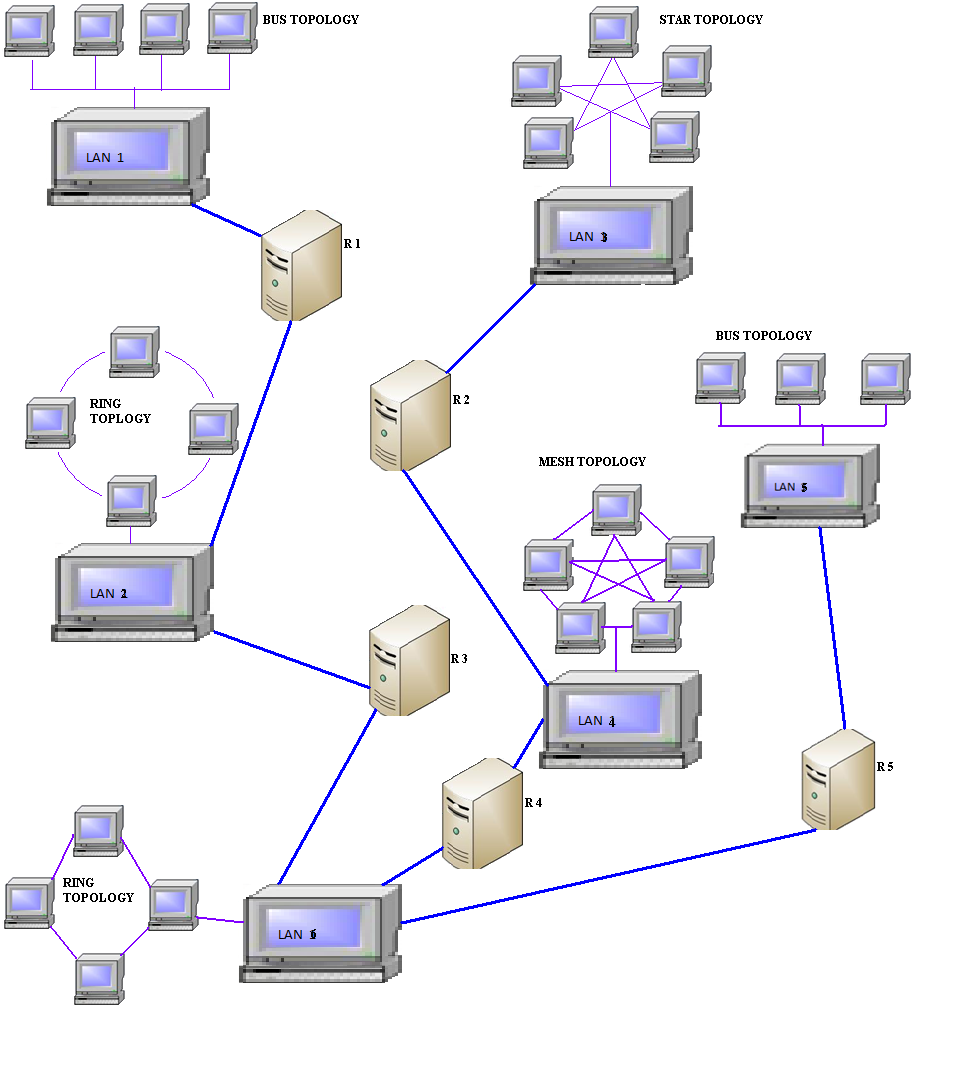
Serversocket(int *port*, int *maxQueue*)-Creates a server socket on the specified *port* with a maximum queue length of *maxQueue*.

ServerSocket(int *port*, int *maxQueue*, InetAddress *localAddress*)-Creates a server socket on the specified *port* with a maximum queue length of *maxQueue*. On a multihomed host, *localAddress* specifies the IP address to which this socket binds.

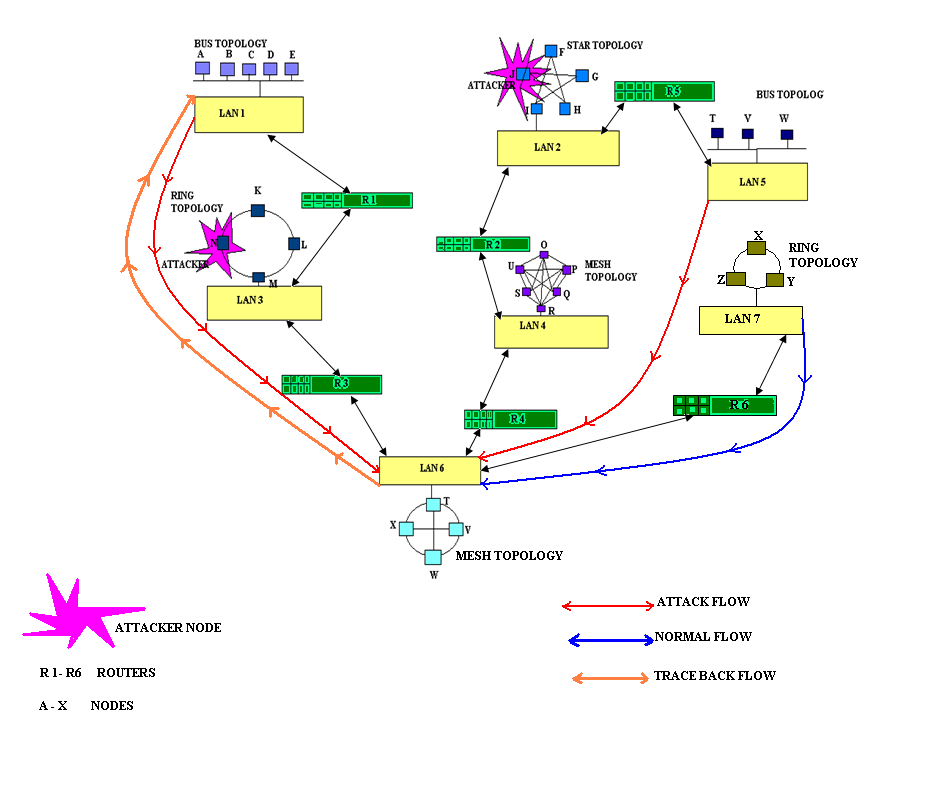
ServerSocket has a method called accept( ), which is a blocking call that will wait for a client to initiate communications, and then return with a normal Socket that is then used for communication with the client

**SYSTEM DESIGN 6. SYSTEM DESIGN**

* 1. **SYSTEM ARCHITECTURE DIAGRAM**



* 1. **DATA FLOW DIAGRAM**

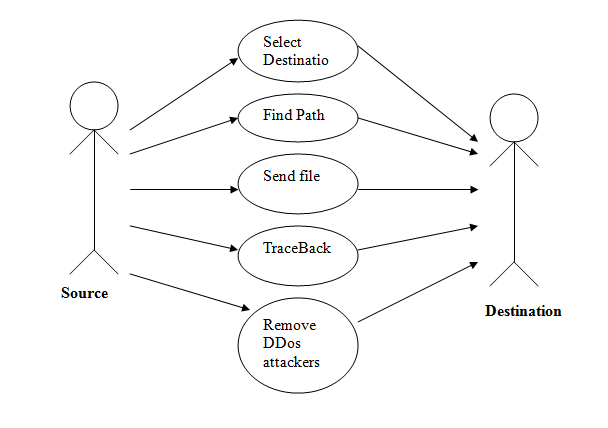


**6.3. UML DIAGRAMS**

**USECASE DIAGARAM**

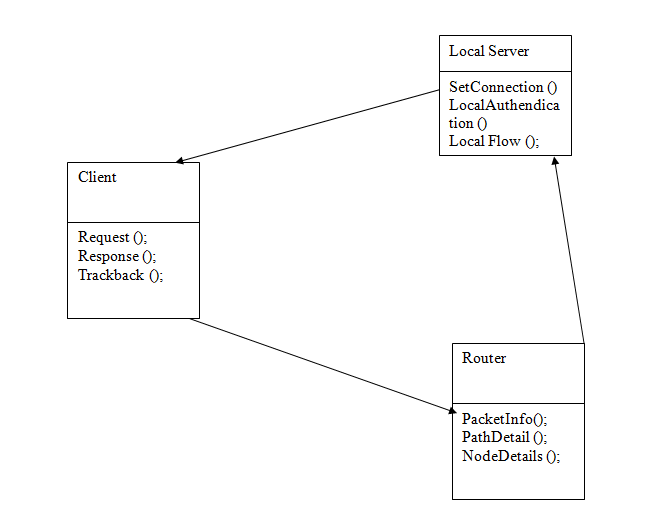
Use case diagrams overview the usage requirements for a system.  They are useful for presentations to management and/or project stakeholders, but for actual development you will find that [use cases](http://www.agilemodeling.com/artifacts/systemUseCase.htm) provide significantly more value because they describe "the meat" of the actual requirements.

**Use cases**. A use case describes a sequence of actions that provide something of measurable value to an actor and is drawn as a horizontal ellipse



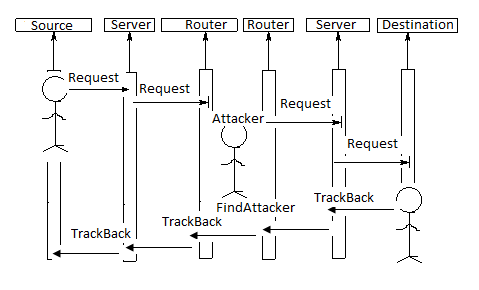
**CLASS DIAGRAM**

Class diagrams are the mainstay of object-oriented analysis and design. class diagrams show the classes of the system, their interrelationships (including inheritance, aggregation, and association), and the operations and attributes of the classes. Class diagrams are used for a wide variety of purposes, including both conceptual/domain modeling and detailed design modeling.



**SEQUENCE DIAGRAM**

Sequence diagrams model the flow of logic within your system in a visual manner, enabling you both to document and validate your logic, and are commonly used for both analysis and design purposes.  Sequence diagrams are the most popular UML artifact for dynamic modeling, which focuses on identifying the behavior within your system.



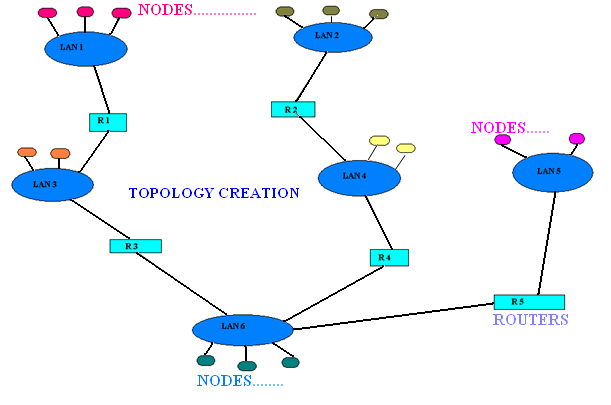
**MODULES DESCRIPTION**

1. **MODULES DESCRIPTION**

* Topology Creation
* DDoS attack
* Entropy variation
* IP Traceback

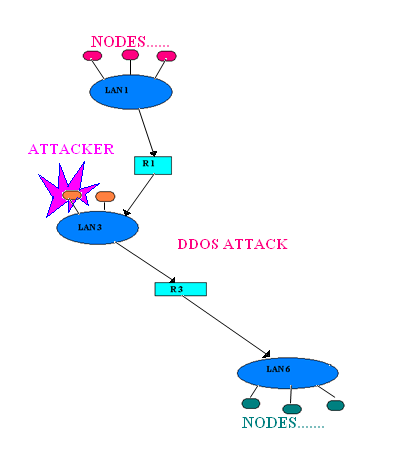
**TOPOLOGY CREATION**

In this module is used to construct the topology. Create the Unstructured Network Connection for Transfers the Data in Different Network for Communication Create the Number of Local Area Network Topology for Local communication. Create the Router, in which Router maintains all Transferring Client Details and this Transferring Data and their Packets. Create Different Nodes, Server and Router in Proper Name, IP Address and Port Number for Data communication. The node is added to give the name of the node, system number and port address of that node. If the node name and port address is already available means to display the message box "Node already Add", otherwise to display the message box "Node add successfully". If the entire node adds successfully to display the node connection frames.



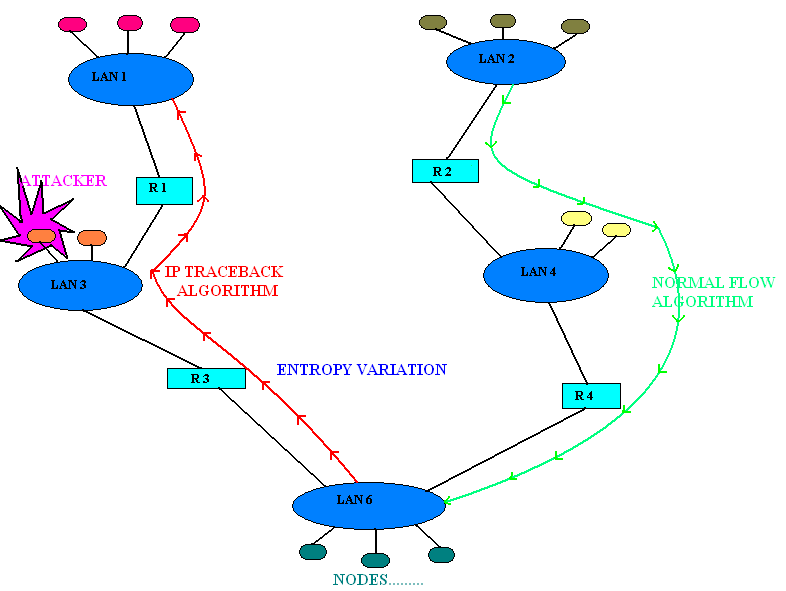
**DDOS ATTACK**

The attacker(s) first establishes a network of computers that will be used to generate the huge volume of traffic needed to deny services to legitimate users of the victim. The attackers discover vulnerable hosts on the network. The next step for the attacker is to install new programs (known as attack tools) on the compromised hosts of the attack network. The hosts running these attack tools are known as zombies, numerous zombies together form an army or botnet. There are two categories of DDoS attacks, Typical DDoS attacks and Distributed Reflection Denial-of-Service (DRDoS) attacks. In a typical DDoS attack, the master computer orders the zombies to run the attack tools to send huge volume of packets to the victim, to exhaust the victim’s resources. Unlike the typical DDoS attacks, the army of a DRDoS attack consists of master zombies, slave zombies, and reflectors. The difference in this type of attack is that slave zombies are led by master zombies to send a stream of packets with the victim’s IP address as the source IP address to other uninfected machines (known as reflectors), exhorting these machines to connect with the victim. Then the reflectors send the victim a great volume of traffic, as a reply to its exhortation for the opening of a new connection, because they believe that the victim was the host that asked for it.



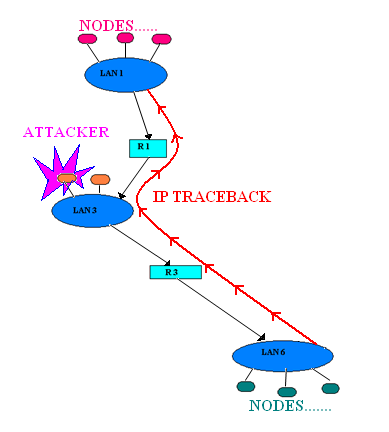
**ENTROPY VARIATIONS**

Compared with the non-attack scenario the upper bound of entropy variation drops when DDos attack flows are passing through a local router. Compared with the non-attack situation, the entropy of a local router drops dramatically when attack flows are passing the local router. For a local router on an attack path, the entropy variation of the output flows is not greater than the summation of the entropy variation of the incoming flows. The entropy variation drops when a local router is closer to the victim and vice versa Based on the partial information of the attack that the traceback algorithms has accumulated, we can estimate the number of zombies to be traced and the maximum length to the most far away zombies. There are no attackers at the upstream routers if a local router’s entropy variation is reasonable. The number of flows for a given router is stable at both the attack cases and nonattack cases. There are two algorithms in the proposed traceback suite, the local flow monitoring algorithm and the IP traceback algorithm. The local flow monitoring algorithm is running at the nonattack period, accumulating information from normal network flows, and progressing the mean and the standard variation of flows. The progressing suspends when a DDoS attack is ongoing. The IP traceback algorithm is installed at routers. It is initiated by the victim, and at the upstream routers, it is triggered by the IP traceback requests from the victim or the downstream routers which are on the attack path.



**IP TRACEBACK**

Once a DDoS attack has been confirmed by any of the existing DDoS detection algorithms, then the victim starts the IP traceback algorithm. The authors proposed the node sampling algorithm, which records the router address to the packet with probability, p, on the routers of the attack path. Then, the probability of a packet marked by a router d that hops away from the victim Based on the number of marked packets, we can reconstruct the attack path. However, it requires large number of packets to improve the accuracy of the attack path reconstruction. Therefore, an edge sampling algorithm was proposed to mark the start router address and end router address of an attack link and the distance between the two ends. The victim will collect all the marked packets from the routers and reconstruct the attack tree based on the traffic rates of the different routers. This trace back method heavily depends on the queuing model. The randomize-and-link approach to implement IP traceback based on the probabilistic packet marking mechanism was proposed. The algorithm targets two aspects: to reconstruct the marks from the marker efficiently and to make the PPM more secure against hackers’ pollution. Two hybrid schemes that combine the packet marking and packet logging method to traceback the attack sources are proposed—Distributed Link-List Traceback (DLLT) and the Probabilistic Pipelined Packet Marking (PPPM).



**SOFTWARE TESTING**

**8. SOFTWARE TESTING**

The purpose of testing is to discover errors. Testing is the process of trying to discover every conceivable fault or weakness in a work product. It provides a way to check the functionality of components, sub assemblies, assemblies and/or a finished product it is the process of exercising software with the intent of ensuring that the Software system meets its requirements and user expectations and does not fail in an unacceptable manner. There are various types of test. Each test type addresses a specific testing requirement.

**MAINTENANCE**

**Correction**

Correct defects in the software: defects relating to incorrect requirements, or incorrectly specifications; defects from any of the construction phases - `bugs'.

**Adaption**

Adapt to changes in the original software and hardware platform. E.g. simpler: MS-DOS to Windows. Complex: stand-alone to client-server.

**Enhancement**

Customer identifies additional requirements.

**Prevention**

After many sets of changes, the software `deteriorates', or otherwise becomes difficult to maintain. See **Reengineering**, **Legacy Systems**.

#### Problems:

* Programmers do not like doing maintenance.
* Maintenance needs a project of its own. It is very uncommon to include maintenance in a (development) contract.
* Cost estimation. Normal software cost estimation is difficult enough; most estimation models do not include maintenance.
* Design deficiencies make system impossible to extend. See **Object-oriented ...**.
* Deterioration of legacy systems.

**8.1 UNIT TESTING**

The first test in the development process is the unit test. The source code is normally divided into modules, which in turn are divided into smaller units called units. These units have specific behavior. The test done on these units of code is called unit test. Unit test depends upon the language on which the project is developed. Unit tests ensure that each unique path of the project performs accurately to the documented specifications and contains clearly defined inputs and expected results.

* 1. **INTEGRATION TESTING**

Testing in which software components, hardware components, or both together are combined and tested to evaluate interactions between them.  
Integration testing takes as its input modules that have been checked out by unit testing, groups them in larger aggregates, applies tests defined in an Integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

* 1. **REGRESSION TESTING**

Regression testing is the process of testing changes to computer programs to make sure that the older programming still works with the new changes. Regression testing is a normal part of the program development process. Test department coders develop code test scenarios and exercises that will test new units of code after they have been written. Before a new version of a software product is released, the old test cases are run against the new version to make sure that all the old capabilities still work. The reason they might not work because changing or adding new code to a program can easily introduce errors into code that is not intended to be changed. It is a quality control measure to ensure that the newly modified code still complies with its specified requirements and that unmodified code has not been affected by the maintenance .

* 1. **ACCEPTANCE TESTING**

Acceptance testing is formal testing conducted to determine whether a system satisfies its acceptance criteria and thus whether the customer should accept the system.

The main types of software testing are:

* Component.
* Interface.
* System.
* Acceptance.
* Release.

Acceptance Testing checks the system against the "Requirements". It is similar to systems testing in that the whole system is checked but the important difference is the change in focus:

Systems testing checks that the system that was specified has been delivered.

Acceptance Testing checks that the system delivers what was requested. The customer, and not the developer should always do acceptance testing. The customer knows what is required from the system to achieve value in the business and is the only person qualified to make that judgment. The User Acceptance Test Plan will vary from system to system but, in general, the testing should be planned in order to provide a realistic and adequate exposure of the system to all reasonably expected events. The testing can be based upon the User Requirements Specification to which the system should conform.

**BLACK BOX TESTING** – Internal system design is not considered in this type of testing. Tests are based on requirements and functionality.

**WHITE BOX TESTING** – This testing is based on knowledge of the internal logic of an application’s code. Also known as Glass box Testing. Internal software and code working should be known for this type of testing. Tests are based on coverage of code statements, branches, paths, conditions.

**INCREMENTAL INTEGRATION TESTING** – Bottom up approach for testing i.e continuous testing of an application as new functionality is added; Application functionality and modules should be independent enough to test separately. done by programmers or by testers.

**FUNCTIONAL TESTING** – This type of testing ignores the internal parts and focus on the output is as per requirement or not. Black-box type testing geared to functional requirements of an application.

**SYSTEM TESTING** – Entire system is tested as per the requirements. Black-box type testing that is based on overall requirements specifications, covers all combined parts of a system.

**END-TO-END TESTING** – Similar to system testing, involves testing of a complete application environment in a situation that mimics real-world use, such as interacting with a database, using network communications, or interacting with other hardware, applications, or systems if appropriate.

**SANITY TESTING** - Testing to determine if a new software version is performing well enough to accept it for a major testing effort. If application is crashing for initial use then system is not stable enough for further testing and build or application is assigned to fix.

**REGRESSION TESTING** – Testing the application as a whole for the modification in any module or functionality. Difficult to cover all the system in regression testing so typically automation tools are used for these testing types.

**LOAD TESTING** – Its a performance testing to check system behavior under load. Testing an application under heavy loads, such as testing of a web site under a range of loads to determine at what point the system’s response time degrades or fails.

**STRESS TESTING** – System is stressed beyond its specifications to check how and when it fails. Performed under heavy load like putting large number beyond storage capacity, complex database queries, continuous input to system or database load.

**PERFORMANCE TESTING** – Term often used interchangeably with ‘stresses and ‘load’ testing. To check whether system meets performance requirements. Used different performance and load tools to do this.

**USABILITY TESTING** – User-friendliness check. Application flow is tested, Can new user understand the application easily, Proper help documented whenever user stuck at any point. Basically system navigation is checked in this testing.

**INSTALL/UNINSTALL TESTING** - Tested for full, partial, or upgrade install/uninstall processes on different operating systems under different hardware, software environment.

**RECOVERY TESTING** – Testing how well a system recovers from crashes, hardware failures, or other catastrophic problems.

**SECURITY TESTING** – Can system be penetrated by any hacking way. Testing how well the system protects against unauthorized internal or external access. Checked if system, database is safe from external attacks.

**COMPATIBILITY TESTING** – Testing how well software performs in a particular hardware/software/operating system/network environment and different combination s of above.

**COMPARISON TESTING** – Comparison of product strengths and weaknesses with previous versions or other similar products.

**ALPHA TESTING** – In house virtual user environment can be created for this type of testing. Testing is done at the end of development. Still minor design changes may be made as a result of such testing.

**BETA TESTING** – Testing typically done by end-users or others. Final testing before releasing application for commercial purpose.

**CONCLUSION**

**9. CONCLUSION**

In this paper, we proposed an effective and efficient IP traceback scheme against DDoS attacks based on entropy variations. It is a fundamentally different traceback mechanism from the currently adopted packet marking strategies. Many of the available work on IP traceback depend on packet marking, either probabilistic packet marking or deterministic packet marking. Because of the vulnerability of the Internet, the packet marking mechanism suffers a number of serious drawbacks: lack of scalability; vulnerability to packet pollution from hackers and extraordinary challenge on storage space at victims or intermediate routers. On the other hand, the proposed method needs no marking on packets, and therefore, avoids the inherent shortcomings of packet marking mechanisms. It employs the features that are out of the control of hackers to conduct IP traceback. We observe and store short-term information of flow entropy variations at routers. Once a DDoS attack has been identified by the victim via detection algorithms, the victim then initiates the pushback tracing procedure. The traceback algorithm first identifies its upstream routers where the attack flows came from, and then submits the traceback requests to the related upstream routers. This procedure continues until the most far away zombies are identified or when it reaches the discrimination limitation of DDoS attack flows. Extensive experiments and simulations have been conducted, and the results demonstrate that the proposed mechanism works very well in terms of effectiveness and efficiency. Compared with previous works, the proposed strategy can traceback fast in larger scale attack networks. It can traceback to the most far away zombies within 25 seconds in the worst case under the condition of thousands of zombies. Moreover, the proposed model can work as an independent software module with current routing software. This makes it a feasible and easy to be implemented solution for the current Internet.

**FUTURE ENHANCMENT**

**10. FUTURE ENHANCEMENT**

In this paper, we proposed an effective and efficient IP traceback scheme against DDoS attacks based on entropy variations. It is a fundamentally different traceback mechanism from the currently adopted packet marking strategies. Many of the available work on IP traceback depend on packet marking, either probabilistic packet marking or deterministic packet marking. Because of the vulnerability of the Internet, the packet marking mechanism suffers a number of serious drawbacks: lack of scalability; vulnerability to packet pollution from hackers and extraordinary challenge on storage space at victims or intermediate routers. On the other hand, the proposed method needs no marking on packets, and therefore, avoids the inherent shortcomings of packet marking mechanisms. It employs the features that are out of the control of hackers to conduct IP traceback. We observe and store short-term information of flow entropy variations at routers. Once a DDoS attack has been identified by the victim via detection algorithms, the victim then initiates the pushback tracing procedure. The traceback algorithm first identifies its upstream routers where the attack flows came from, and then submits the traceback requests to the related upstream routers. This procedure continues until the most far away zombies are identified or when it reaches the discrimination limitation of DDoS attack flows. Extensive experiments and simulations have been conducted, and the results demonstrate that the proposed mechanism works very well in terms of effectiveness and efficiency. Compared with previous works, the proposed strategy can traceback fast in larger scale attack networks. It can traceback to the most far away zombies within 25 seconds in the worst case under the condition of thousands of zombies. Moreover, the proposed model can work as an independent software module with current routing software. This makes it a feasible and easy to be implemented solution for the current Internet.

Future work could be carried out in the following promising directions:

* 1. The metric for DDoS attack flows could be further explored. The proposed method deals with the packet flooding type of attacks perfectly. However, for the attacks with small number attack packet rates, e.g., if the attack strength is less than seven times of the strength of nonattack flows, then the current metric cannot discriminate it. Therefore, a metric of finer granularity is required to deal with such situations.
  2. Location estimation of attackers with partial information. When the attack strength is less than seven times of the normal flow packet rate, the proposed method cannot succeed at the moment. However, we can detect the attack with the information that we have accumulated so far using traditional methods, e.g., the hidden Markov chain model, or recently developed tools, e.g., the network tomography. We have a strong interest to explore this for the whole attack diagram.
  3. Differentiation of the DDoS attacks and flash crowds. In this paper, we did not consider this issue—the proposed method may treat flash crowd as a DDoS attack, and therefore, resulting in false positive alarms. We have a high interest to explore this issue.

**REFRENCES**

**11. REFERENCES**

* “IP Flow-Based Technology,” ArborNetworks, http://www.arbornetworks.com, 2010.
* C. Patrikakis, M. Masikos, and O. Zouraraki, “Distributed Denial of Service Attacks,” The Internet Protocol J., vol. 7, no. 4, pp. 13-35, 2004.
* T. Peng, C. Leckie, and K. Ramamohanarao, “Survey of Network- Based Defense Mechanisms Countering the DoS and DDoS Problems,” ACM Computing Surveys, vol. 39, no.1, p. 3, 2007.
* Y. Kim et al., “PacketScore: A Statistics-Based Packet Filtering Scheme against Distributed Denial-of-Service Attacks,” IEEE Trans. Dependable and Secure Computing, vol. 3, no. 2, pp. 141-155, Apr.-June 2006.
* H. Wang, C. Jin, and K.G. Shin, “Defense against Spoofed IP Traffic Using Hop-Count Filtering,” IEEE/ACM Trans. Networking, vol. 15, no. 1, pp. 40-53, Feb. 2007.
* Y. Chen and K. Hwang, “Collaborative Detection and Filtering of Shrew DDoS Attacks Using Spectral Analysis, ” J. Parallel and Distributed Computing, vol. 66, pp. 1137-1151, 2006.
* K. Lu et al., “Robust and Efficient Detection of DDoS Attacks for Large-Scale Internet,” Computer Networks, vol. 51, no. 9, pp. 5036- 5056, 2007.
* R.R. Kompella, S. Singh, and G. Varghese, “On Scalable Attack Detection in the Network,” IEEE/ACM Trans. Networking, vol. 15, no. 1, pp. 14-25, Feb. 2007.
* P.E. Ayres et al., “ALPi: A DDoS Defense System for High-Speed Networks,” IEEE J. Selected Areas Comm., vol. 24, no. 10, pp. 1864-1876, Oct. 2006.
* R. Chen, J. Park, and R. Marchany, “A Divide-and-Conquer Strategy for Thwarting Distributed Denial-of-Service Attacks,”IEEE Trans. Parallel and Distributed Systems, vol. 18, no. 5, pp. 577-588, May 2007.
* Yaar, A. Perrig, and D. Song, “StackPi: New Packet Marking and Filtering Mechanisms for DDoS and IP Spoofing Defense,”IEEE J. Selected Areas Comm., vol. 24, no. 10, pp. 1853-1863, Oct.2006.
* Bremler-Bar and H. Levy, “Spoofing Prevention Method,” Proc. IEEE INFOCOM, pp. 536- 547, 2005.
* J. Xu and W. Lee, “Sustaining Availability of Web Services under Distributed Denial of Services Attacks,” IEEE Trans. Computers, vol. 52, no. 2, pp. 195-208, Feb. 2003.
* W. Feng, E. Kaiser, and A. Luu, “Design and Implementation of Network Puzzles,” Proc. IEEE INFOCOM, pp. 2372-2382, 2005.
* X. Yang, D. Wetherall, and T. Anderson, “A DoS-Limiting Network Architecture,” Proc. ACM SIGCOMM, pp. 241-252, 2005.

**APPENDICES 12. APPENDICES**

**12.1. SAMPLE CODING:**

//package Login;

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

import javax.swing.event.\*;

import java.util.\*;

public class ClientRegister

{

JLabel l1,l2,l3,l4;

JTextField t1,t2,t3;

JButton b1,b2;

JComboBox cb1;

Container panel1;

Vector cli=new Vector();

JFrame f=new JFrame();

public ClientRegister()

{

initialization1();

try

{

}

catch (Exception e1)

{

System.out.println(e1);

System.out.println("end");

}

}

public void initialization1()

{

l1=new JLabel("Client Name");

l2=new JLabel("Client IPAddress");

l3=new JLabel("Client PortNumber");

l4=new JLabel("LAN Name");

t1=new JTextField();

t2=new JTextField();

t3=new JTextField();

b1=new JButton("Create");

b2=new JButton("Exit");

cli.add("list");

cli.add("list");

cli.add("list");

cb1 = new JComboBox(cli);

f=new JFrame();

panel1=(JPanel)f.getContentPane();

panel1.setBackground(new Color(132,128,150));

f.setLayout(null);

addComponent(panel1, l1, 50,50,150,50);

addComponent(panel1, l2, 50,100,150,50);

addComponent(panel1, l3, 50,150,150,50);

addComponent(panel1, l4, 50,200,150,50);

addComponent(panel1, t1, 350,60,150,30);

addComponent(panel1, t2, 350,110,150,30);

addComponent(panel1, t3, 350,160,150,30);

addComponent(panel1, cb1, 350,210,150,30);

addComponent(panel1, b1, 150,350,100,40);

addComponent(panel1, b2, 250,350,100,40);

b1.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e)

{

b1\_actionPerformed(e);

}

});

b2.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e)

{

b2\_actionPerformed(e);

}

});

f.setTitle("ClientCreation");

f.setLocation(new Point(100, 100));

f.setSize(new Dimension(600,500));

f.setDefaultCloseOperation(WindowConstants.EXIT\_ON\_CLOSE);

f.setVisible(true);

}

private void addComponent(Container container,Component c,int x,int y,int width,int height)

{

c.setBounds(x,y,width,height);

container.add(c);

}

private void b1\_actionPerformed(ActionEvent e)

{

try

{

}

catch(Exception ex)

{

ex.printStackTrace();

}

}

private void b2\_actionPerformed(ActionEvent e)

{

try

{

}

catch(Exception ex)

{

ex.printStackTrace();

}

}

public static void main(String args[])

{

try

{

ClientRegister reg=new ClientRegister();

}

catch(Exception e)

{

System.out.println("expc"+e.toString());

}

}

}

//package Login;

import java.awt.Dimension;

import java.awt.Point;

import java.awt.event.\*;

import java.awt.\*;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.\*;

import javax.swing.event.\*;

import java.net.\*;

public class NCRegistration

{

JLabel l1,l2,l3,l4,l5;

JTextField t1,t2,t3;

JButton b1,b2;

Container panel1;

Socket socket;

JFrame f=new JFrame();

String strSRep;

String PName,IPAdd,sPort,strReq,strRe,strRep;

public static ServerConnection sc =new ServerConnection();

public NCRegistration()

public void initialization1()

{

l1=new JLabel("Topology Initiation");

l2=new JLabel("Router Number");

l3=new JLabel("Server Number");

b1=new JButton("Submit");

b2=new JButton("Exit");

// l5.setIcon(new ImageIcon("bg.bmp"));

f=new JFrame();

panel1=(JPanel)f.getContentPane();

f.setLayout(null);

addComponent(panel1, l1, 130,30,250,50);

addComponent(panel1, l2, 10,90,150,20);

addComponent(panel1, t1, 200,90,150,30);

addComponent(panel1, l3, 10,130,100,20);

addComponent(panel1, t2, 200,130,150,30);

addComponent(panel1, b1, 100,210,100,40);

addComponent(panel1, b2, 200,210,100,40);

//addComponent(panel1, l5, 0,0,404,300);

b2.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e)

{

b2\_actionPerformed(e);

}

});

});

f.setTitle("Topology Initiation");

f.setLocation(new Point(300, 100));

f.setSize(new Dimension(404, 300));

f.setDefaultCloseOperation(WindowConstants.EXIT\_ON\_CLOSE);

f.setVisible(true);

}

private void addComponent(Container container,Component c,int x,int y,int width,int height)

{

c.setBounds(x,y,width,height);

container.add(c);

}

private void b2\_actionPerformed(ActionEvent e)

{

System.exit(0);

}

private void b1\_actionPerformed(ActionEvent e)

{

try

{

PName=t1.getText().trim();

IPAdd=t2.getText().trim();

sPort=t3.getText().trim();

int RR = Integer.parseInt(PName);

int SR = Integer.parseInt(IPAdd);

int CR = Integer.parseInt(sPort);

int ch;

strReq="NCRCegistration@"+PName+"@"+IPAdd+"@"+sPort;

socket=new Socket("localhost",8585);

ObjectOutputStream OutStream=new ObjectOutputStream(socket.getOutputStream());

OutStream.writeObject(strReq);

/\*ObjectInputStream InStream=new ObjectInputStream(socket.getInputStream());

strSRep=(String)InStream.readObject();

String strSp[]=strSRep.split("&");

String strcValue=strSp[0];

String strRPeer=strSp[1];

if(strcValue.equals("true"))

{\*/

JOptionPane.showMessageDialog(null," Successfully Stored ","Information Message",JOptionPane.INFORMATION\_MESSAGE);

f.dispose();

new RRegistration(RR,SR,CR);

/\*}

else

{

JOptionPane.showMessageDialog(null," Peer Details Not in DataBase","Warning Message",JOptionPane.INFORMATION\_MESSAGE);

}\*/

}

catch(Exception ex)

{

ex.printStackTrace();

}

}

public static void main(String args[])

{

NCRegistration reg=new NCRegistration();

}

}

//package Login;

import java.awt.Dimension;

import java.awt.Point;

import java.awt.event.\*;

import java.awt.\*;

import javax.swing.\*;

import java.sql.\*;

import javax.swing.\*;

import java.io.\*;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.\*;

import javax.swing.event.\*;

import java.net.\*;

import java.util.\*;

public class ContstructNetwork

{

JLabel l1,l2,l3,l4,l5;

JTextField t1,t2,t3;

JButton b1,b2,b3;

Container panel1;

Socket socket;

JList lis1,lis2,lis3,lis4;

JScrollPane sb1,sb2,sb3,sb4;

Vector RSCRouterList=new Vector();

Vector RSCServerList=new Vector();

Vector RSCClientList=new Vector();

Vector RSCConstructList=new Vector();

Vector RSCRouterList1=new Vector();

Vector RSCServerList1=new Vector();

Vector RSCClientList1=new Vector();

int Rv;

int Sv;

int Cv;

Vector v=new Vector();

JFrame f=new JFrame();

String strSRep;

String PName,IPAdd,sPort,strReq,strRe,strRep;

public static ServerConnection sc =new ServerConnection();

public ContstructNetwork()

{

initialization1();

try

{

try

{

int ch;

strReq="RSCRouterDetails@";

socket=new Socket("localhost",8585);

ObjectOutputStream OutStream1=new ObjectOutputStream(socket.getOutputStream());

OutStream1.writeObject(strReq);

ObjectInputStream InStream2=new ObjectInputStream(socket.getInputStream());

RSCRouterList=(Vector)InStream2.readObject();

lis1.setListData(RSCRouterList);

ObjectOutputStream OutStream3=new ObjectOutputStream(socket.getOutputStream());

OutStream3.writeObject(strReq);

ObjectInputStream InStream4=new ObjectInputStream(socket.getInputStream());

RSCServerList=(Vector)InStream4.readObject();

lis2.setListData(RSCServerList);

ObjectOutputStream OutStream5=new ObjectOutputStream(socket.getOutputStream());

OutStream5.writeObject(strReq);

ObjectInputStream InStream6=new ObjectInputStream(socket.getInputStream());

RSCClientList=(Vector)InStream6.readObject();

lis3.setListData(RSCClientList);

}

catch(Exception ex)

{

ex.printStackTrace();

}

}

catch (Exception e1)

{

System.out.println(e1);

System.out.println("end");

}

}

public void initialization1()

{

l1=new JLabel("Topology Construction");

l2=new JLabel("RouterName");

l3=new JLabel("ServerName");

l4=new JLabel("ClientName");

l5=new JLabel("Constructing Groups");

t1=new JTextField();

t2=new JTextField();

t3=new JTextField();

b1=new JButton("Construct");

b2=new JButton("Finish");

b3=new JButton("Exit");

lis1=new JList();

lis2=new JList();

lis3=new JList();

lis4=new JList();

sb1=new JScrollPane();

sb2=new JScrollPane();

sb3=new JScrollPane();

sb4=new JScrollPane();

sb1.setViewportView(lis1);

sb2.setViewportView(lis2);

sb3.setViewportView(lis3);

sb4.setViewportView(lis4);

// l5.setIcon(new ImageIcon("bg.bmp"));

f=new JFrame();

panel1=(JPanel)f.getContentPane();

f.setLayout(null);

addComponent(panel1, l1,330,30,250,50);

addComponent(panel1, l2, 50,90,150,20);

addComponent(panel1, sb1, 20,130,150,230);

addComponent(panel1, l3, 250,90,150,20);

addComponent(panel1, sb2, 210,130,150,230);

addComponent(panel1, l4, 430,90,150,20);

addComponent(panel1, sb3, 390,130,150,230);

addComponent(panel1, l5, 580,90,150,20);

addComponent(panel1, sb4, 570,130,150,230);

b3.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e)

{

b3\_actionPerformed(e);

}

});

b2.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e)

{

b2\_actionPerformed(e);

}

});

b1.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e)

{

b1\_actionPerformed(e);

}

});

lis1.addListSelectionListener(new ListSelectionListener() {

public void valueChanged(ListSelectionEvent e1)

{

if(!e1.getValueIsAdjusting())

{

try

{

Object o=lis1.getSelectedValue();

//String Ro = o.toString();

//String RRo="Router-->"+Ro;

if(Rv==0)

{

v.add(o);

RSCRouterList.remove(o);

else

{

JOptionPane.showMessageDialog(null," Router Already Present","Information Message",JOptionPane.INFORMATION\_MESSAGE);

}

}

catch (Exception es)

{

System.out.println(es);

}

}

}

});

lis2.addListSelectionListener(new ListSelectionListener() {

public void valueChanged(ListSelectionEvent e1)

{

if(!e1.getValueIsAdjusting())

{

Object o=lis2.getSelectedValue();

//String So = o.toString();

//String SSo="Server-->"+So;

if(Sv==0)

{

v.add(o);

RSCServerList1.add(o);

RSCServerList.remove(o);

lis4.setListData(v);

lis2.setListData(RSCServerList);

Sv=1;

}

else

{

JOptionPane.showMessageDialog(null," Server Already Present","Information Message",JOptionPane.INFORMATION\_MESSAGE);

}

}

}

});

lis3.addListSelectionListener(new ListSelectionListener() {

public void valueChanged(ListSelectionEvent e1)

{

if(!e1.getValueIsAdjusting())

{

RSCClientList.remove(o);

lis4.setListData(v);

lis3.setListData(RSCClientList);

Cv++;

}

}

});

lis4.addListSelectionListener(new ListSelectionListener() {

public void valueChanged(ListSelectionEvent e1)

{

if(!e1.getValueIsAdjusting())

{

try

{

//if(o.equals(o))

//{

//System.out.print("Router");

//}

if(RSCRouterList1.contains(o))

{

RSCRouterList.add(o);

lis1.setListData(RSCRouterList);

RSCRouterList1.remove(o);

Rv=0;

}

if(RSCServerList1.contains(o))

{

}

if(RSCClientList1.contains(o))

{

RSCClientList.add(o);

lis3.setListData(RSCClientList);

RSCClientList1.remove(o);

Cv--;

}

v.remove(o);

lis4.setListData(v);

}

catch (Exception et)

{

System.out.println(et);

}

}

}

});

f.setDefaultCloseOperation(WindowConstants.EXIT\_ON\_CLOSE);

f.setVisible(true);

}

private void addComponent(Container container,Component c,int x,int y,int width,int height)

{

c.setBounds(x,y,width,height);

container.add(c);

}

private void b2\_actionPerformed(ActionEvent e)

{

try

{

strReq="FinishRct@";

socket=new Socket("localhost",8585);

ObjectOutputStream OutStream=new

System.exit(0);

}

catch (Exception ts)

{

System.out.println(ts);

}

}

private void b3\_actionPerformed(ActionEvent e)

{

System.exit(0);

}

private void b1\_actionPerformed(ActionEvent e)

{

try

{

if(Rv!=0&&Sv!=0&&Cv!=0)

{

System.out.println("clientdetails"+csize);

int xr;

String clientdetails="";

for(xr=0;xr<=csize-1;xr++,xr++)

{

String srg1=(String)(RSCClientList1.get(xr));

System.out.println("clientdetails"+srg1);

//if(!srg1.contains("null"))

//{

clientdetails+=srg1+"@";

//}

}

System.out.println(clientdetails);

ObjectOutputStream OutStream=new ObjectOutputStream(socket.getOutputStream());

OutStream.writeObject(strReq);

//ObjectInputStream InStream=new ObjectInputStream(socket.getInputStream());

// RSCRouterList=(Vector)InStream.readObject();

//lis1.setListData(RSCRouterList);

v.clear();

}

else

{

JOptionPane.showMessageDialog(null," Its is not a Proper Structure","Information Message",JOptionPane.INFORMATION\_MESSAGE);

}

}

catch(Exception ex)

{

ex.printStackTrace();

}

}

public static void main(String args[])

{

ContstructNetwork reg=new ContstructNetwork();

}

}

**12.2. SAMPLE SCREENSHOTS:**

