Detection of denial of service attacks

The subject of the thesis has been approved by the department council of the Department of Information Technology on 16.01.2008.

Examiners: Professor Esa Kerttula
Senior assistant Pekka Jäppinen
Supervisor: M.Sc. Oleg Chistokhvalov

Lappeenranta 27.01.2008

Dmitrii Karasev
Karankokatu 4 A 4
53810 Lappeenranta
Finland
Abstract

Lappeenranta University of Technology
Department of Information Technology

Dmitrii Karasev

Detection of denial of service attacks

Thesis for the degree of Master of Science in Technology
2007

57 pages, 22 figures and 2 appendices

Examiners: Professor Esa Kerttula
Senior assistant Pekka Jäppinen

Keywords: denial of service, distributed denial of service attack, detection, CUSUM algorithm, ns-2 network simulator.

This thesis studies techniques used for detection of distributed denial of service attacks which during last decade became one of the most serious network security threats. To evaluate different detection algorithms and further improve them we need to test their performance under conditions as close to real-life situations as possible. Currently the only feasible solution for large-scale tests is the simulated environment. The thesis describes implementation of recursive non-parametric CUSUM algorithm for detection of distributed denial of service attacks in ns-2 network simulator – a standard de-facto for network simulation.
# Table of contents

Terms and abbreviations

1. Introduction

2. Denial of service attacks
   2.1 Terminology
   2.2 Attack techniques
      2.2.1 Bandwidth depletion
      2.2.2 Resource depletion
   2.3 Impact

3. Countermeasures
   3.1 Prevention
   3.2 Detection
   3.3 Mitigation

4. CUSUM algorithm implementation
   4.1 CUSUM algorithm
   4.2 Code implementation in ns-2

5. Conclusion

References

Appendix A

Appendix B
Terms and abbreviations

DoS - Denial of Service
DDoS - Distributed Denial of Service
Ns-2 - Network simulator
CUSUM - Cumulative Sum
SYN - TCP synchronization message
FIN - TCP closing connection message
SYN+ACK - TCP acknowledgement and synchronization message
ACK - TCP acknowledgement message
PUSH+ACK - TCP acknowledgement and forced data delivery message
AS - Autonomous System
ISP – Internet Service Provider
IDS - Intrusion Detection System
IPS - Intrusion Prevention System
TTL - Time-To-Live
D-WARD - DDoS Network Attack Recognition and Defense
NOMAD – Traffic-based network-monitoring framework for anomaly detection
MULTOPS – MUlti-Level Tree for Online Packet Statistics
OTcl – Object-oriented Tcl
1. Introduction

Nowadays businesses rely more and more in their practices/business models on their online presence. Some of the companies have proceeds from the online store as their one and only revenue source, for others “being on the Net” is still just a means to provide no more than technical support and feedback for their customers and to advertise their products. Nevertheless of the degree they depend on their online divisions it is still very frustrating and in most cases costly (especially for web-driven enterprises) to have online services disrupted. It could happen for a number of reasons, just to name a few: hardware or software failure, virus infiltration, insider abuse and especially denial of service (DoS) attacks. The latter term covers numerous techniques which deny the legitimate user access to the online service. Among them currently the utmost importance is given to distributed denial of service (DDoS) attacks.

DDoS attacks came to the attention of the research community about nine years ago [1], when in August of 1999 a DDoS tool called Trinoo was spread across at least 227 systems and as a result of their activities a single University of Minnesota host was unavailable for two days. From that moment on DDoS attacks became quite commonplace occurrence. Soon they changed their status from being just a nuisance to a serious threat, both in terms of direct financial losses from disruption of services to legitimate clients in the event of an attack and associated expenditures needed to maintain the infrastructure to prevent, detect and mitigate attacks.

Widely publicized events of February 2000 proved the seriousness of this threat. On February 7, 2000 Yahoo web site became a target for the DDoS attack – for 3 hours the portal was unavailable. Next day, February 8, Amazon, Buy.com, CNN and eBay were also targeted, which either shut them down completely or considerably slowed their operations. On February 9 it was a turn for ZDNet and E*Trade. During an attack Buy.com could only provide to its clients 9.4 percent availability as opposed to usual 100 percent, and for CNN’s customers this number
dropped to 5 percent, while ZDNet and E*Trade sites went practically offline [2]. Even today, almost 9 years later, this attack still remains probably most singularly costly event in the history of denial of service attacks - “according to the Yankee Group, a Boston consulting firm, the DDoS attack in February cost approximately $1.2 billion” [3].

A key problem when trying to solve DDoS attacks is attack detection which importance cannot be overstated. We need to detect an attack in progress as soon as possible for a number of reasons. First of all, the sooner the attack is detected prior to inflicting any real damage, the more time the system under the attack has to implement some defense measures. Second, detection of the attack usually ascertains also the identity of those systems which participate in it. This data is potentially useful for taking the legal action and prosecuting the guilty party. Third, if the attack can be detected close to its sources, corresponding filtering mechanism can be turned on, dropping attack flows and preventing the bandwidth waste. Of course, all those opportunities are available only if a given detection mechanism is really doing what it is supposed to do, if it is really effective.

In this regard, it becomes crucial to be able to test different detection approaches not only on the basis of their theoretical performance but also under conditions as close to different real situations as possible. Considering available options, we have three venues to pursue. First, we can run detection scenarios on test beds, which are usually of limited size and thus any results are of limited credibility. Second, we can use collected network traces and run our tested detection engines against them. Of course, there are almost no publicly available network traces, so we have to collect them by ourselves – a rather time consuming and tedious process. Then we have to take into account the place where the network trace is going to be collected - it is one thing to collect statistics on a number of backbone nodes and completely another if we are limited only to our own local network traffic. Consider also that during time the nature of traffic changes. For example, during late 80s and early 90s majority of the Internet traffic was either FTP or e-mail. During late 90s it was all about Web traffic. Nowadays a substantial amount of traffic is due to peer-to-peer applications. Third, we can use network simulators which allow verification of
scenarios to be run and are easily scalable to hundreds and possibly thousands of
nodes, allowing us to test complex topologies with different kinds of traffic flows.

The last option is probably the only solution if we are aiming our research to be independently verified and consequently accepted by academic community. As a standard de-facto in network simulation is ns-2 network simulator [4], we are obviously interested in implementing any detection mechanisms in ns-2. Currently, though, there is no publicly available code, neither in main ns-2 distribution, nor in code contributed by users, which performs any detection of denial of service attacks.

The detection mechanism which in my view merits foremost the outright implementation in the code for ns-2 simulator is non-parametric Cumulative Sum (CUSUM) algorithm, which is simple and efficient. The CUSUM algorithm was shown to be optimal in terms of detection accuracy for parametric model and has good performance for non-parametric model [5]. The efficiency of such an algorithm was evaluated and validated on the sets of collected network traces [6-10]. As such implementation of CUSUM algorithm in the ns-2 would be of some value for researchers interested in detection of DDoS attacks in a simulated networking environment. The CUSUM algorithm due to its low computational overhead is also a good candidate for implementation at routers of Internet Service Providers (ISP) as it allows real-time attack detection. Prior to deployment of CUSUM algorithm in real life environment it would be beneficial to run the ns-2 simulation with the use of CUSUM detection algorithm to allow for tweaking of the algorithm parameters (to incorporate some of the real-life onsite conditions) – so the ISP operators can also benefit from the use of proposed ns-2 code.

My contribution in this paper is the actual code implementation of recursive non-parametric CUSUM algorithm for the ns-2 simulator which uses as its statistic measure the number of new network addresses, detected at a leaf router, where this mechanism is implemented/installled. The justification for the validity of the choice of the statistic used is the observation in [11], where it was shown that during DDoS attacks most source IP addresses are new to a victim.

The rest of the paper is organized as follows. Chapter 2 introduces main concepts used in denial of service research, such as terminology used, accepted
taxonomy and typical ways in which denial of service attacks are launched. The impact of denial of service attacks on businesses is also briefly considered.

Chapter 3 presents some known countermeasures against DDoS attacks. They can be put into three separate groups, like attack prevention and preemption, attack detection and filtering, attack source traceback and identification. Some of the examples from each of those groups are given.

Chapter 4 presents CUSUM algorithm, used in our implementation and discusses its realization in the ns-2 network simulator.

Chapter 5 offers conclusions we were able to gather from the network simulations using CUSUM algorithm.

2. Denial of service attacks

2.1 Terminology

The denial of service (DoS) attacks can be divided into two groups: vulnerability attacks and flooding attacks. First one, as the name suggests, is based on exploiting the vulnerabilities of a target machine/host or target application. These vulnerabilities are typically either a software bug in the implementation or a bug in the configuration of a service. The attacker sends to the target host a few messages in specific format which uses this given vulnerability. As a programmer did not foresee the used particular input, then the application either considerably slows down or goes into infinite loop or freezes or completely crashes the whole system. One of the classical examples of such an attack is a “ping-of-death” also known as “long ICMP” attack, which gained popularity in 1996 – 1997 [12]. The ping request usually consists of 64 bytes (84 bytes including IP header). As most systems cannot handle an IP packet with size more than its allowed maximum size of 65,536 bytes, then upon receiving a packet of a size more than that a buffer overflow could happen, which quite often leads to a system crash. Eventually, such a bug was fixed (it was
done in late 1997 by issuing corresponding patches for different operating systems),
and now “ping-of-death” is no more than the historical curiosity.

Obviously, this kind of attacks depends heavily on previously undiscovered weaknesses in network protocols/services and their implementations in software. As soon as such a weakness is found, the system which runs corresponding application is open to the attack. That’s why vulnerability attacks are hard to predict, as we do not know what can go wrong.

Second group, flooding attacks (known also as saturation attacks) are designed to exhaust some of the key resources at the target, like bandwidth, CPU time, memory, etc. For example, complex messages need more CPU cycles to process, lengthy messages take up bandwidth and messages/requests for new connections use up buffer memory. As soon as all the resources are tied up, the legitimate users cannot access the service, i.e. they are denied it. The key here is that flooding attacks rely not on content but particularly on volume. As we as an attacker start with only one host generating numerous requests to the target host, executing an attack, this situation is described as DoS – denial of service attack. To bring down the target we would need to generate hundreds and thousands of packets per second to saturate the resource. As such, this attack is easily identified and dealt with. So instead we launch our attack from numerous hosts (preferably servers), distributing the generation of packets to the target host, thus the name DDoS – distributed denial of service attack. Every packet stream from one of those hosts is aggregated at the target so we have an amplification of traffic.

Apart from a greater amplification factor there are other advantages to DDoS attacks, at least from the point a view of an attacker. Usually a server machine has more processing power, memory and especially bandwidth than a client machine (a workstation). So using server machines the attacker has better chances of saturating a target. Then there is a matter of stopping the attack. If the attack comes from one single source and it is possible to trace it back, then in most cases it will be possible to stop it only if a source system owner/administrator manually does the action. If we have 1000 attack sources, then we would need to contact 1000 different system
administrators to stop them. And getting thousands of people to do something is an overwhelming challenge.

In order to perform DDoS attack from numerous hosts first we need to gain access to them. As such those compromised hosts are often called the “secondary victims/targets” and the host/system/network under attack itself is called the “primary victim/target”. The use of secondary targets allows attackers to use much larger base of packet generating hosts while providing higher degree of anonymity as real flooding attack is performed by secondary sources so tracking down a real attacker becomes a formidable task. Another benefit for the attacker, as was mentioned earlier, is that the aggregation of attack traffic is done only at the target so we can restrict our packet generation rates to much lower values and it becomes very hard to distinguish this kind of traffic at the source networks and intermediate nodes on the way to the target. As DDoS attacks are much more disruptive, currently the majority of research on denial of service attacks is done with emphasis on distributed systems.

The Figure 1 provides an illustration on how the denial of service attack is performed [13].

To some extent the modus operandi of an attacker can be compared to the way a military strategist plans a campaign. First off we need to establish a base of operations. We start from an attacker’s computer usually referred to as a client – a system from which we communicate with the rest of the DDoS attack system. Then we begin scanning the Internet in search of hosts with vulnerabilities we can explore, i.e. hosts with nonexistent or weak security. When such a host is found we install on this system a handler – software package which is used to further control DDoS attack system. On other compromised hosts we install agents – somewhat simpler software packages which are going to be used to carry out an attack against a target. Agents are also alternatively referred to as zombies. Handlers are used to identify which agents are active, when to launch the attack, when to update agents and so on, basically performing control functions. Agents, though, are responsible only for actual attack traffic generation.
Typically the owners of compromised systems have no knowledge of the presence of agents. Those agents can be configured to be in connection either with a single handler or numerous ones. And finally, we launch an attack from agents, flooding the target system, thus using greater amplification factor of DDoS. Once the attack is in progress we can take the handler offline, to make it impossible to trace the attack back to us. To make it harder to distinguish attack traffic from legitimate communications, handlers are installed at routers or network servers with high volume of traffic thus masking messages between the client and the handler, and between the handler and agents. This scenario is usually referred to as “agent-handler model”. Its variation is “IRC-based model” where instead of using handlers the communication between an attacker and agents is done using IRC (Internet Relay Chat) channel [14].
2.2 Attack techniques

To better understand and describe existing denial of service flooding attacks, different taxonomies were proposed [14-17]. One of these taxonomy schemes is presented in Figure 2 [15].

In this scheme terms “bandwidth depletion” and “resource depletion” cover the whole range of flooding attacks with emphasis, correspondingly, to exhaustion of available bandwidth and exhaustion of particular resource at the target.

![Figure 2. Taxonomy of DDoS attacks [15]](image)

2.2.1 Bandwidth depletion

Flooding attacks saturate the target by sending numerous packets using either ICMP or UDP packets. The packets are sent by agents directly to the target flooding the target’s network bandwidth with IP traffic. For UDP flood attacks an attacker
could send packets either to predetermined or random port, sometimes spoofing the IP source address and thus masking the identity of agents. ICMP attacks are waged by sending large volumes of ping requests to the target.

During an amplification attack an attacker, i.e. agents send requests to the third-party network broadcast addresses. Upon receiving such broadcast messages, a router sends this broadcast message to all the IP addresses inside the broadcast address range. As the source IP address of the packet is spoofed, giving the IP address of the target, that is where replies to those requests are sent back, significantly amplifying the number of messages to the target. An attacker can sometimes, instead of establishing agents first, send broadcast messages to affected networks directly. In this way nodes inside the broadcast network act as agents themselves, so the attacker does not need to infiltrate any hosts and install any agent software. Smurf and fraggle attacks are representatives of amplification attacks.

Smurf attacks appeared first around 1997 [18]. During smurf attack (named after the program used to launch such an attack) an attacker sends ICMP ping echo request to networks broadcast addresses, giving spoofed IP address of a target. If routers of affected networks allow forwarding of broadcast messages, then this request is sent to all the nodes inside the affected network and nodes’ echo reply would be sent to the target. For the class /24 network amplification factor would be a couple of hundreds and for the class /16 network amplification factor would be several thousands. There is, though, an easy prevention mechanism, which includes configuring routers not to respond to ping requests to broadcast messages and not to forward them.

Fraggle attack is a modification of a smurf attack [19]. The difference is that it uses UDP packets instead of ICMP. An attacker sends UDP echo packets to the networks broadcast addresses using in most of the cases destination port number 7, which is an echo port. Sometimes, though, traffic is directed to the port 19 - character generation port (chargen) with spoofed target echo port as the source port, creating an infinite loop.
2.2.2 Resource depletion

Resource depletion attacks (as in Figure 2) use some particular feature of the network protocol or packet format, which is susceptible either to misuse or erroneous input. Exploiting the vulnerability leads to the resource exhaustion and consequently to the denial of service to the legitimate users.

TCP SYN (also known as SYN flood) attacks are based on misusing TCP protocol when establishing initial connection [20]. An attacker sends to the target stream of TCP SYN packets with spoofed source addresses. Connection establishment in TCP protocol requires three-way handshake. Upon receiving SYN request the target replies issuing SYN+ACK message and reserving an entry in the connection queue. As the destination address for this SYN+ACK message was spoofed, this packet is sent to non-existent or incorrect address and the final part of the three-way process never completes as there is no concluding ACK message. An entry for the half-open connection remains in the queue until it expires, usually in about one minute. We have the connection queue of limited size, so by issuing high volume of bogus SYN packets, it can be filled up pretty quickly and there would be no place in it for incoming SYN messages from the legitimate or otherwise users – a typical denial of service situation.

PUSH+ACK attack uses active ACK and PUSH flags (set to one) in the TCP packet header. When the target receives such a packet it is a signal to force the data delivery without waiting for buffers to fill and to send back an acknowledgment. When the attack rate is high enough, the target system cannot process large amount of incoming packets and quite possible will crash down.

Malformed packet attack branch (Figure 2) of the taxonomy tree describes situations when an attacker deliberately messes up the packets sent to the target. IP address attack uses the same source and destination addresses in sent messages aiming to confuse the receiving side and to crash the target’s operating system. IP packet options attack sets all service quality bits in the packet header to one; as a
result the target’s system has to use additional processing time to analyze the traffic. If the system under such an attack has to process a great amount of modified packets, it can eventually waste off all the processing capacity.

2.3 Impact

DoS attacks have been happening under different guises for decades. Consider, for example, famous Morris worm, which coincidentally is considered to be the first worm [21]. Robert T. Morris, a student at Cornell University, in November of 1988 released a program, which explored some of the vulnerabilities of Unix sendmail, Finger, rsh/exec and weak passwords. It was supposed to infiltrate affected hosts by guessing passwords and then to replicate itself. Unfortunately due to programming error the multiplication rate of the worm was excessive and it quickly spread, managing to infect up to 6,000 Unix machines. Those hosts had so many copies of the worm running simultaneously that the infected systems slowed down to the point of being unusable. In addition many systems were disconnected from the network by local administrators to avoid being infected. Overall effect was that the whole Internet was effectively brought down. The cost of damages due to the denial of service was estimated to be in the range of $10 - 100 million. Besides being first of its kind in impact it had on the Internet community, this incident brought to the public awareness the issues of security and reliability of the Internet.

DDoS attacks, though being much newer have proved to be much more disruptive and financially damaging. One has to take into account, of course, that during 1988 the size of the Internet was up only to 60,000 hosts [22], primarily at academic and research institutions, so any disruption in service, however significant in scope, had almost no effect on businesses. But during last decade there was unprecedented growth in the number of hosts, which in the middle of 2007 reached a number of 489 million hosts [23]. For many businesses the ability to rely on stability
of their online services became the basis of their very own existence. Consequently, any disruption of online service would be very expensive in terms of lost profit and marred public image.

In terms of occurrence denial of service attacks are certainly some of the most frequent among all misuse attacks, as witnessed by yearly surveys of CSI (Computer Security Institute) (Figure 3) [24]. As can be seen from this picture, the ratio of respondents who have experienced such attacks during last 8 years never falls below 25%. There is, of course, a question of how truthfully those surveys represent the reality, i.e. how prevalent denial of service attacks are in the Internet. Obviously it would be a sensible assumption to presume (especially in business community) that not all attacks are reported in media either because they were successfully repelled or, if they achieved their goal to cause some disruption of services, the companies were too embarrassed to report it in view of a belief that admitting it is going to tarnish their reputation.

Unfortunately, apart from voluntarily answering survey questions there is almost no way to collect statistical data on denial of service attacks, especially worldwide. Service and content providers are very reluctant to give away this kind of information considering it to be of a private nature, assuming, of course, that they are collecting corresponding data. Then, even if they were in a cooperative mood, it is still a very serious logistical challenge to monitor passing traffic at enough sites to get a feeling/statistic for Internet-wide attacks.

There is, though, one paper on the subject [25], which tries to evaluate globally happening denial of service attacks. Here the authors analyzed during three years of observations (2001 - 2004) 22 distinct network traces. They found that we have on average 2,000 – 3,000 active denial of service attacks per week worldwide. These numbers mean that denial of service presents a clear and ongoing problem and we need to research it in order to design adequate countermeasures.
Figure 3. Types of attacks of misuse during last 12 month, by percent of respondents [24]
3. Countermeasures

On the face of it designing and implementing effective countermeasures against saturation denial of service attacks seems to be a simple and straightforward task. We certainly understand how denial of service attacks occur and we are well aware of not so many variations of attacks existing at the moment. So why are those attacks still present? And as it usually happens, for this simple question there is quite a complicated answer, or at least part of it.

First off, let us consider the situation from the point of view of an attacker.

There are many tools available for organizing, configuring, managing and launching of attacks, such as trinoo, stacheldraht, mstream and others [26]. These tools are very simple to use and have a lot of automated features, which makes them ideal tool at the hands of so called “script kiddies” of which a substantial part probably does not even understand what they are doing.

Apart from the cases when we are using specifically crafted messages (worms and viruses) attack traffic is very similar in content to the legitimate traffic and distinguishing them is a real challenge.

The very nature of distributed attacks makes some of the proposed defense techniques at least impractical. Consider, for example, situation where there are 10,000 or 20,000 agents flooding the target, using spoofed addresses. Little good does it to us to determine the identity of those agents. We still have to contact owners of compromised hosts to stop the agents. Given the sheer numbers involved, the task seems rather infeasible. Take into account also, that the continuing growth in the number of hosts with its fair share of inexperienced users gives us no indication that the potential pool of agents, i.e. systems with poor security, will ever diminish in near future.

Existing network protocols have no measures in place to distinguish between legitimate and spoofed network source addresses.

Second, from the point of view of a security specialist, there are certain challenges, both technical and social.
DDOS attacks require naturally also a distributed response. It means that we cannot completely rely on any defense mechanism implemented only at the target. We need a distributed, probably coordinated, response system. It also has to cover as many points in the Internet as possible to be able to deal with different agents. As currently there is no way to enforce introduction of such a system on Internet-wide basis, many researchers do not even investigate distributed solutions, believing that their research would be confined only to simulations and at utmost to some test beds. As such any observations made in these unrealistic environments are not very credible.

Distributed solutions require, as was pointed out, wide-spread coordinated distributed deployment. Unless there is a proven benefit (like reduced economic risk or increased profits from charging customers more for additional security) businesses are reluctant to pay for those solutions and unfortunately this situation will probably not change much in near future.

Having made this quite disheartening conclusion, it is necessary to point out that there is still a lot of research in the field. Granted, that the major interest shifted now to local solutions - either detection or mitigation techniques, let us consider some of the stages involved in dealing with denial of service problem.

Usually there are three distinguished steps: attack prevention (before the attack), attack detection and filtering (during the attack) and attack source traceback and identification (during and after the attack) [27].

### 3.1 Prevention

Best prevention would be to try to stop attacks from happening in the first place. As the majority of attacks use spoofed IP source addresses that’s where we may concentrate our efforts. There are a number of techniques proposed, which include:
• Ingress filtering. A router is set up to block the packets with not-valid IP source addresses from propagating further upstream [28]. Usually this level of filtering is done at the ISP, to which a number of networks are connected. Every packet arriving to the router from any particular network is examined and if its source address does not match a domain prefix of a network connected to the ingress filter, than it gets dropped.

• Egress filtering. As opposed to ingress, egress filtering deals with packets leaving a network on the way upstream, allowing only packets with assigned or allocated IP addresses to pass through [29]. Essentially ingress and egress filtering serve the same purpose, namely to stop spoofed traffic from leaving your network. Difference is only in the placement of those filters. Interesting thing about this kind of filtering is that it does not protect your own network from denial of service attacks, merely making it harder for the attacker to launch an attack from your infected hosts. In a sense, implementing egress filtering is a social activity, being “a good neighbor” to other networks.

• Route-based distributed filtering. Instead of relying on ingress filtering to prevent spoofed traffic from entering the Internet, we can implement route-based distributed packet filtering (DPF) at the routers of autonomous systems (AS). Those filters use the route information to find out whether the packet arriving at a router – like a border router of an AS – is valid in regard to its own source/destination addresses [30]. Even with partial coverage - as small as 18% - of all autonomous systems in the Internet a synergistic filtering effect is achieved and the spoofed IP flows are prevented from reaching other autonomous systems. Even if we cannot block all the spoofed traffic, due to the complexity of Internet topology, the flows are sparse enough for us to be able to pinpoint their real origin within a small, constant number of sites (less than 5 for Internet AS topologies)

• History-based IP filtering. [31] According to this approach an edge router maintains the history of all the legitimate IP addresses which previously appeared in the system. When the system becomes overwhelmed with requests only those connections which are on the list are allowed to pass. This scheme does not need multi-host implementation, requires little configuration and serves multiple
types of traffic, as the only thing that matters is source IP address. At the same time, if an attacker knows about this kind of defense, he/she still can manage to circumvent it by injecting prior to the attack some of its low-intensity flows into legitimate traffic, so IP addresses of attack hosts will be recorded into the IP address database. This trick can be prevented by increasing the time interval during which IP addresses must appear to qualify as frequent.

In addition to filtering there are some other steps we can undertake to prevent happenings of denial of service attacks. First, we can turn off/disable those network services which misuse can be potentially harmful. For example, as fraggle attacks use echo or character generator services, disabling them will help to defend against those attacks. Second, as attacker depends on vulnerabilities/security flaws, we can limit his ability to infect systems with agents by applying latest security patches. Third, when the system comes under attack, we can switch targets’s IP address to a new one – technique known as moving target defense [3]. As most denial of service attacks are based on IP addresses, as soon as we change target’s IP address, the attack in progress loses its focus. Internet routers must be informed about the change and edge routers will drop the attack flows. Of course, attackers can by adding a domain name service tracing function to the DDoS attack tools render this kind of defense rather superficial.

3.2 Detection

Our ability to withstand the denial of service attack is greatly affected by our ability to correctly identify the presence of attack traffic and to filter it out. The whole process of detection and filtering is also greatly influenced by our choice for a monitoring point. Certainly, when we deploy any kind of detection mechanism at a target’s network, it is relatively easy to identify attack traffic as it aggregates there. On the other hand, the sheer volume of that traffic makes filtering, allowing legitimate traffic and dropping or rate-limiting attack traffic, a very complicated and
process-consuming matter, as usually we need to process every single packet. For example Paul Holbrook, Director for Internet Technologies for CNN said regarding the famous denial of service attack in February of 2000: "In our case, what caused us trouble was not that we weren't filtering it out. We were filtering it, but the problem was that the routers were so busy filtering that that in itself compromised us. The routers still have to process each packet. The cure was putting the filter on a bigger router outside of our site" [32].

As shown in Figure 4 [27], we can put our monitoring point at four different locations. The first, most obvious one, right at the target’s network, provides higher degree of confidentiality in determining denial of service activity, while effectiveness of packet filtering is rather low as we have to cope with large volumes of normal and attack traffic throne together.

![Figure 4. Possible locations for performing detection and filtering [27]](image)

On the opposite end is the location at attack source networks, where attack flows originate. Unless there are a relatively large number of nodes participating in the attack, we have a small chance of detecting any attack activity, but if we do manage to detect attack flows, then it is more effective to filter them out at the source. The effectiveness of packet filtering declines rapidly on the packet route to destination, because more normal packets would also be dropped, as no filtering
technique is 100% accurate. Two other locations – target’s ISP network and further upstream networks – represent compromises/tradeoffs between effectiveness of packet filtering and effectiveness of attack detection.

Generally the effectiveness of any detection mechanism can be rated by such parameters as detection time, false positive ratio (FPR) and false negative ratio (FNR). The false positive ratio is the number of packets classified as attack packets (positive) by our detection engine that our found to be normal (negative), divided by the total number of confirmed normal packets. The false positive ratio is the number of packets classified as normal (negative) by detection engine that are confirmed to be attack packets (positive), divided by the total number of confirmed attack packets. For the effective detection mechanism these ratios should be as low as possible. For the measure of filtering effectiveness there is the normal packet survival ratio (NPSR) which is a percentage of normal packets that still get through to the target while that target is under denial of service attack. Higher NPSR values correspond, obviously, to higher ratios of normal traffic unaffected by the attack [27].

Designing detection mechanisms it is necessary to implement not only the ability to distinguish between normal and attack traffic, but also the ability to make distinction between attack traffic and traffic associated with flash events. Flash event is defined as “a large surge in traffic to a particular Web site causing a dramatic increase in server load and putting severe strain on the network links leading to the server, which results in considerable increase in packet loss and congestion” [11]. A typical example of a flash event would be a rush-in of users to the major news site, such as CNN or BBC in the case of major terrorist attack, like September 11. In this case we would expect those sites to be swamped with numerous requests.

The key difference between a flash event and denial of service attack is that in the former case the requests are legitimate but this observation does not help unless we are aware of some other distinctions in behavior of both events. In [11] an analysis of a number of network traces was done which revealed quite interesting conclusion, namely that a Web site receives many more requests from previously seen clusters, i.e. subnets/local networks during a flash event than during an attack. Important thing here is that we compare number of clusters and not the number of
individual clients, as the percentage of individual clients is small in all events. From the common sense it seems that legitimate requests are from those clusters where at least some of the users have already visited the Web site while in the case of denial of service attacks addresses are random – as the attacker was able to install agents at some randomly chosen hosts.

As we are interested particularly in detection of denial of service attacks, let us consider some of the existing solutions. Usually detection techniques can be put into one of two distinct groups: DDoS attack-specific detection which obviously is based on the some signatures exhibited by attack flows and anomaly-based detection which, as the name suggests, observes normal traffic and monitors it for any abnormalities which would naturally be a sign of an attack.

The attack-specific detection relies on a number of signatures to be found in the passing traffic – detection engine is constantly trying to match observed traffic to a signatures database. The main advantage of such approach is that the known attacks (for which we have signatures) are easily and reliably identifiable. On the other hand, to be effective the database has to be kept up-to-date and it is not possible to recognize new attacks. There are currently a number of popular solutions for monitoring traffic which use signatures to detect denial of service attacks. As detection is based on signatures, those solutions are typically a part of much broader security suits. Two of the examples are proprietary IBM Proventia Network Intrusion Prevention System (IPS) [33] and open source Intrusion Detection System (IDS) Snort [34].

Anomaly-detection systems can be further divided into separate categories, such as statistical analysis based and rate limiting based.

Statistical analysis based:

- NOMAD[35] is one of the oldest mechanisms which relies on statistical analysis of IP packet header information, such as time-to-live (TTL), source/destination address and packet length as also as router’s timestamps. Detection algorithms use path changes, flows shift and packet delay variance to detect traffic overload, router misconfiguration/malfunction and component failure.
As this paper was published in 1999, there is no explicit mention in it of denial of service attacks.

- A correlation between traffic patterns at the target and traffic patterns at the source is a basis for another approach [36]. This scheme uses time series analysis on the data collected from routers into Management Information Base (MIB) to detect statistical patterns.

- Sequential change-point detection. Algorithms of this group use the change in statistic during the denial of service attack as an indication of attack happening. The theoretical foundations are presented in [37] and [5]. The idea behind this approach is that any rapid change in the traffic during an attack results in the corresponding change of statistical models, thus a deviation from "normal model" is a sign of an attack. We collect any related statistic at our observation point during fixed time periods. The objective of sequential change-point detection is to decide if observed time series remain statistically consistent and if not, to determine time moment at which change occurred. As the object of research is Internet traffic, for which we do not have simple parametric model, as Internet traffic is of complex stochastic nature, instead we rely on the non-parametric CUSUM (Cumulative Sum) method for an attack detection [5]. A description of CUSUM algorithm is presented in section 4.1. Detection schemes, which use CUSUM algorithm for DDoS attack detection, employ as an observation statistic either number of new IP addresses at the observation point or in the case of detection SYN flooding attacks the number of SYN-FIN pairs [9, 10].

Rate limiting based:

- One of the representatives of this group is D-WARD - DDoS Network Attack Recognition and Defense. [38] This approach is based on the assumption that denial of service attacks have to be stopped as close to the source as possible. D-WARD is installed at the source router in the deployment network. The observation component of the detection engine monitors all packets passing through the router and collects statistics on two-way communications between sets of local addresses and the rest of the Internet. Periodically statistics are compared to the legitimate traffic models and the decision is made whether flows and connections are legitimate
or not. The observation component passes the information to the rate-limiting component, which in turn makes a decision to set, modify or remove the rate limit on the flows. Rate-limit rules are passed on to the traffic-policing component, which uses a list of classified-legitimate connections to enforce the rules on the allegedly attack traffic, trimming the outgoing flows.

- Another example of detection technique based on rate limiting is MULTOPS [39]. MULTOPS uses the assumption that the rates of incoming and outgoing connections under normal conditions should be in balance. Under this scheme network devices maintain a multilevel tree, keeping track of certain network statistics and store results in nodes corresponding to subnet prefixes at different aggregation levels. As soon as the attack happens, there is a disproportionate difference in rates of incoming and outgoing traffic for certain flows and as a response those flows are rate limited. As opposed to D-WARD, MULTOPS can be implemented either at source side or target side. Though by design D-WARD and MULTOPS are quite similar, there are some differences. For example, MULTOPS imposes fixed, non-selective limit on incoming or outgoing traffic as opposed to D-WARD which uses selective and dynamic response. Unfortunately, because of the use of a disparity between in and out flows as a key parameter, both approaches cannot prevent proportional attacks, i.e. attacks with balanced rates of incoming and outgoing connections.

3.3 Mitigation

Mitigation techniques – actions we need to take after an attack - consist primarily of approaches dealing with tracing back the attack flows to their sources and identifying the real attackers. Also an essential part of mitigation stage consists of blocking attack traffic. As any automated response system will only worsen the situation in the event of a false alarm, the blocking part is usually done under manual supervision by contacting upstream router administrators.
Currently there are a number of different solutions to the problem of tracing back the attack, because this is a very difficult problem. The degree of complexity is emphasized by two observations: first, due to the original design of IP protocol, the source address of any packet can be very easily spoofed; second, because routers operate on the stateless basis, they only know next hop for the forwarded packet instead of being aware of a whole route from the source to the destination. As the complete overhaul of existing TCP/IP protocol to include source accountability seems very unlikely, we are left with only roundabout means to traceback the source of the attacks. At its basic form, an administrator of the system under attack will call his ISP to ask from which direction the packets are coming. It looks like a very tedious and inefficient process, so a number of automated systems were proposed:

- ICMP traceback [40] (also called iTrace) with low probability (1 out of 20,000) sends new ICMP packet containing Traceback message (which includes previous hop, next hop, timestamp, variable-length traced packet and authentication) along with forwarding packets to its destination. If we assume that enough messages are received at the destination, then the origin of traffic can be reconstructed by a chain of Traceback messages. There are, though, some disadvantages to this scheme. First off, extra traffic is generated for those Traceback messages. Second, ICMP traffic can be a subject of filtering or other limitations as opposed to the normal traffic. Third, due to the distributed nature of the attack, it is questionable if we would be able to collect all the Traceback messages. Consider the situation, for example, when routers closer to agents have lower probability to send to the target corresponding Traceback messages as the overall amount of traffic from any agent is relatively low. On the other hand, routers closer to the target, due to aggregation of traffic, will have higher probability of sending the right Traceback message to the target. An improvement of the ICMP traceback scheme, which uses intention-bit in the routing and forwarding table, is called Intention-Driven ICMP Traceback [41]

- Probabilistic packet marking [42] randomly encodes partial route path information and includes the traceback data into IP packets. This mechanism can be applied during or after the attack and it does not require any additional network traffic, router storage or packet size increase. On the other hand, this mechanism
lacks backward compatibility, i.e. it uses 16-bit identification field in IP header, usually reserved for identification of IP fragments that belong to different packets. Also it does not support IPv6 and requires modification of Internet routers to generate those marks on the fly.

- Pushback [43] is another mitigation mechanism. It is built around the idea of notifying the upstream router to rate-limit or drop a set of traffic identified as poor (aggregate). In the Aggregate-based Congestion Control a set is defined as aggregate if it has identifiable property. In this approach a pushback daemon decides whether or not an attack is in progress by running a detection algorithm. There are some advantages and disadvantages of this mechanism: incremental deployment is possible and as such there is no need for upstream routers. On the other hand, there is big storage requirement for pushback daemon to keep track of analyzed packets. And finally, there is a lack of trust relationship between ISPs to accept pushback requests from others.

4. CUSUM algorithm implementation

As was pointed out in section 3.2, CUSUM algorithm which is used for detecting DDoS attacks, is one of the variations of sequential change-point statistical approach. A number of research papers are published on the subject, giving detailed analysis of its performance [6-10, 44]. It was determined through this research that this algorithm provides simple and robust detecting mechanism which is characterized by small computational overhead. Because the non-parametric CUSUM algorithm is used, we do not need to know any of the parameters for the statistical model, which again proves CUSUM algorithm to be quite solid in performance.
4.1 CUSUM algorithm

CUSUM algorithm in our deployment uses a number of new source network addresses as its statistic. Before we continue with discussion of algorithm’s operation in ns-2 network simulator – actual source code implementation, let us consider the placement of the detection monitoring point. We can position our detection engine either at the first-mile or last-mile leaf routers, which are, basically, the gateways to and from the Internet for the local intranet. Every leaf router can serve in both capacities, depending on the direction of incoming and outgoing traffic. For example, for the traffic leaving the intranet the leaf router is the first-mile router. Otherwise, for the incoming traffic into the intranet the leaf router is the last-mile router. Usually any detection engine is placed on the link connecting the Internet to the local intranet, monitoring the bidirectional traffic.

In ns-2 simulations we are running we use the last-mile router scenario, in which we intercept all incoming traffic. We examine all packets coming to the leaf node, stripping the header and retrieving the source address of any intercepted packet. This operation is done constantly, so we have at any moment an up-to-date database of all observed source network addresses. At fixed time observation intervals $\Delta$ we check this database, comparing it with the database of source addresses, which was recorded at the previous time. From a number of consecutive check-ups we as a result have a random sequence representing the number of new IP addresses in a time interval $\Delta$. To smooth the resulting time series in regard to traffic variations, we normalize those values by dividing them by the overall number of discovered addresses at any given time. Resulting sequence $x_n$ under normal conditions, when the number of newly discovered addresses is relatively small, will have mean value $\alpha$ which is quite close to 0 (as indicated in Figure 5) [6]. As soon as there is an influx of new addresses during an attack, the mean value will experience the step-like minimum increase by value $h$ so the mean value would be now $\alpha + h$.
As one of the assumptions of CUSUM algorithm is that mean value of a random sequence is negative during normal operation, and becomes positive during change, we transform our random sequence $x_n$ into new random sequence $z_n$ (Figure 6) [6]. This transformation is $z_n = x_n - \beta$, where $\beta$ is such a constant that $a = \alpha - \beta < 0$.

CUSUM algorithm measures the values of the statistic used (in our case it is the number of new addresses) which overlaps the threshold N for the purpose of attack detection, using recursive expression (see Figure 7) [6].

$$y_n = \begin{cases} y_{n-1} + z_n, & y_n > 0 \\ 0 & \text{otherwise} \end{cases}$$

where

$$x^+ = \begin{cases} x, & x > 0 \\ 0, & \text{otherwise} \end{cases}$$
If at some time we have a large value of $y_n$, exceeding threshold $N$ then it is a good indication that indeed our system is under distributed denial of service attack. Suppose, as presented in Figure 7, that the attack begins at time $m$. Then the value of $y_n$ will increase until it reaches the threshold at time $\tau_N$ - a moment when attack is declared and it is time to implement some defensive measures. As can be seen, the operation of this algorithm is rather straightforward and very simple.

4.2 Code implementation in ns-2

Ns-2 was developed at the University of Southern Carolina beginning from 1989 and it can be run on multiple flavors of UNIX operating system and even under Windows [4]. Ns-2 is a discrete event simulator. As such it consists of two major parts. The modules responsible for the actual event processing are written in C++ to provide for the needed high run-time speed. For setting parameters, specifying topologies and simulation configuration it is necessary to use OTcl (Object-oriented Tcl) interpreter - which serves as a frontend to the whole simulator. The CUSUM algorithm is written in C++ with linkage to the corresponding class in OTcl to allow us to create an instance of the structure implementing CUSUM algorithm through interpreter frontend.
In general, due to the complexity of ns-2 (when a packet goes on its way from one node to another through numerous different components), there is a multitude of choices of which objects to modify to incorporate CUSUM algorithm. For example, consider Figure 8 [45], which presents the route that a packet can take in ns-2 between two unicast nodes.

![Figure 8. Ns-2. Packet flow between two unicast nodes [45]](image)

We can write the corresponding code of CUSUM implementation as part of a separate agent (like FTP agent in Figure 8), which sole purpose would be to process packets which pass through the node of interest (leaf router) to the target destination. Then, of course, we would be forced to rewrite address classifier component to allow for this agent to be able to sniff on passing packets. In some regard the situation will be like a snow avalanche: change a small detail and incremental changes you have to make in other components will soon make the whole project well beyond anything feasible. As was pointed earlier, usually detection mechanism is put directly on the link between the leaf router and the intranet. Following this tradition, we implement the detection engine also on the link between the leaf router - a node where the traffic converges and a node which represents a target of the denial of service attack – supposedly one of the nodes of hypothetical intranet. Then the question of which component to modify has an answer. We rewrite a link component which models a
channel between two nodes - the leaf router and the target. As in ns-2 queues are implemented as part of the simplex or duplex link, we incorporate CUSUM algorithm into existing code for the DropTail queuing component.

The whole detection engine consists of three distinct parts: detection component, responsible for determining the presence of an attack by examining the traffic; decision component, which decides whether or not an attack is in progress; filtering component, responsible for dropping those packets which considered to be a part of attack flows.

When we start the simulation using our detection engine, in the beginning there is a certain time interval – we could call it training period (chosen arbitrarily, could be any number - the only condition is that during this time enough packets are received at the leaf router on the way to target to build up the address database). For our simulations the training period of 10 seconds is used. During this time, network addresses of all the packets are put into database and no detection or decision components are activated.

As soon as simulation time exceeds training period, we activate the detection engine, i.e. at the end of each observation time \( \Delta \) we compare collected statistic, i.e. we compare the overall number of all observed addresses for the current database with the same number from previous addresses database. The difference is obviously a number of newly discovered addresses during time interval \( \Delta \). Then we proceed in accordance with algorithm outlined in previous section: the detection component computes the value of \( y_n \). The decision component examines cumulative sum of new addresses and if at some time this value exceeds predetermined threshold, then the attack flag is “up” and update of database of discovered addresses is suspended. Otherwise, if there is no attack detected, we continue with the database update (previous addresses database is replaced with the current one). In the case of attack detection, the filtering component comes into play. It examines every packet and if its source addresses is a match with any of new addresses, then this packet gets dropped in the queue. This way only packets from older, proven to be not attack sources are allowed to pass to the target.
One of the examples of used topologies is presented in Figure 9. Basically all the tried variations are slightly modified versions of it, i.e. there is one leaf router to which the rest of the nodes, which comprise the simulation scenario are connected through other subnodes. A setup of one of the used simulation runs which results are presented later is as follows: a target node #101 is connected to the leaf router #0 through two simplex links, at one of which we use the modified DropTail queue on the way from the leaf router to the target. The size of the particular used simulation is 102 nodes. Legitimate traffic and attack traffic is represented by UDP flows originating from nodes #1 - 100 and passing through leaf router on the way to the target (Figure 9). The simulation running time is set up to 100 seconds.

Obviously, the time interval $\Delta$ greatly influences the sensitivity of the attack detection: the longer this time interval than the longer is the time delay in detecting the attack in progress. At the same time, longer time intervals tend to miss the denial of service attacks with burst structure, as Figure 10 and Figure 11 later show.

Figure 9. Ns-2. Used network topology
As an example of the dependence of sensitivity of our algorithm to the observation time interval there are results of network simulations for the case where the parameters used were: $\beta = 0.1$ and $N = 0.1$ for different observation periods (5 and 2 seconds). It is necessary to note, though, that the use of particular values for the threshold $N$ and the offset $\beta$ here and in the simulation runs results of which discussed later does not justify the blind use of those values for the CUSUM algorithm for any general case. The reason is that we lack a parametric model for $x_n$ and, correspondingly, for $z_n$. As such any discussion of optimal values for $N$ and $\beta$ makes sense only if we consider the onsite conditions, i.e. take into consideration the local network conditions.

The graph of $y_n$ was drawn for the same traffic generators and same topology in both cases. The burst nature of the denial of service attack is modeled arbitrarily by directing first wave of the attack in the interval from 50.3 till 61.2 seconds and the second wave from 72.3 till 80.2 seconds of simulation scenario. The length of both attacks in the range of about 10 seconds was chosen to allow our algorithm to detect the change of measured statistic for the case of observation time of 5 seconds and less.

When observation time interval was 5 seconds (Figure 10), the time delay between the start of the attack at 50.3 seconds and the moment when $y_n$ reaches the threshold value (at 54th second) is 3.7 seconds – detection time. For the case when observation interval was 2 seconds, the detection time is 1.7 seconds ($52 - 50.3$ seconds). Also it can be noted that when the observation period is 5 seconds, the attack flag turns “up” at 54th second and is maintained in this state till the end of the simulation, so the detection engine reports the attack when in reality it has already stopped. On the other hand, smaller observation time period of 2 seconds (Figure 11) resolves the presence of the second wave of the attack.

As we can see, when we have smaller observation time interval, we are able to distinguish finer structure of denial of service traffic, especially its pulse or burst nature. Bigger time intervals seem to mask this attack feature.

Figure 12 and Figure 13 present graphs $y_n$ for the case $\beta = 0.05$ and $N = 0.1$ for same different time intervals of 5 and 2 seconds (network topology and attack
flows are preserved). Comparing pairs of graphs with different values of $\beta$ we can observe that as expected for the smaller $\beta$ the peaks of $y_n$ have bigger values. The less is the offset of $z_n$ into negatives, then consequently more likely is the value of $y_n$ to exceed zero and be counted towards larger value to indicate an attack.

The same kind of observations for the same kind of topology and attack flows was done for the case of $\beta = 0.05$ and $\beta = 0.1$ for $N = 0.2$. Results are presented in Figures 14 - 17. As expected, if we raise the detection threshold, the detection time increases significantly. Detection delay is now about 24 seconds for both cases (for observation time intervals of 5 and 2 seconds) mainly because of the second attack wave. It seems that the first wave was not sufficient enough to raise the value of $y_n$ above the detection threshold.

Another pair of observation of behavior $y_n$ is done for different values of $\beta = 0.1$ and $\beta = 0.05$ for $N = 0.3$. Results are shown in Figures 18 - 21. Evidently, the threshold value of 0.3 was set too high and our detection engine was not able to detect any attack at all regardless of $\beta$, though as shown in Figure 18 the second peak of $y_n$ almost reached the detection threshold.

![Figure 10. $y_n$ vs. time. Threshold $N = 0.1$, $\beta = 0.1$ Observation time interval is 5 seconds.](image-url)
Figure 11. $y_n$ vs. time. Threshold $N = 0.1$, $\beta = 0.1$. Observation time interval is 2 seconds.

Figure 12. $y_n$ vs. time. Threshold $N = 0.1$, $\beta = 0.05$. Observation time interval is 5 seconds.
Figure 13. $y_n$ vs. time. Threshold $N = 0.1$, $\beta = 0.05$. Observation time interval is 2 seconds.

Figure 14. $y_n$ vs. time. Threshold $N = 0.2$, $\beta = 0.05$. Observation time interval is 5 seconds.
Figure 15. $y_n$ vs. time. Threshold $N = 0.2$, $\beta = 0.05$. Observation time interval is 2 seconds.

Figure 16. $y_n$ vs. time. Threshold $N = 0.2$, $\beta = 0.1$. Observation time interval is 5 seconds.
Figure 17. $y_n$ vs. time. Threshold $N = 0.2$, $\beta = 0.1$. Observation time interval is 2 seconds.

Figure 18. $y_n$ vs. time. Threshold $N = 0.3$, $\beta = 0.05$. Observation time interval is 5 seconds.
Figure 19. $y_n$ vs. time. Threshold $N = 0.3$, $\beta = 0.05$. Observation time interval is 2 seconds.

Figure 20. $y_n$ vs. time. Threshold $N = 0.3$, $\beta = 0.1$. Observation time interval is 5 seconds.
One important note to make is that in our CUSUM algorithm implementation we artificially trim the cumulative sum of the sample statistic in the case when during an observation period there are no new network source addresses and the attack flag is in “up” position. The reasoning behind that decision is that we presume that if there are no new addresses recorded during time $\Delta$, there are two possibilities for that. First off, the attack really stopped. As soon as it happens we need to set the attack flag in “down” position, essentially notifying the filtering engine to stop its activity and lessening the CPU load. So if there are no new addresses we assign the value of the threshold to the variable $y_n$ – our statistic but the attack flag is still in “up” position – just a precaution, really. If during next time interval there is still no new addresses discovered or their number is substantially low enough to merit statistic’s value to drop below threshold, then we declare the attack flag “down” – attack has stopped for real. Second, if the absence of new addresses during time $\Delta$ was no more than a fluke, and the attack is still in progress, then at next observation time the statistic value $y_n$ will be easily upset over threshold and the attack flag still going to be kept “up”.

Figure 21. $y_n$ vs. time. Threshold N = 0.3, $\beta = 0.1$. Observation time interval is 2 seconds.
This tweak of the CUSUM algorithm was done to address the feature of the conventional algorithm, briefly discussed in [6] and evident from the recursive expression for $y_n$, namely that the conventional scheme can not accurately detect the end of the attack because the CUSUM algorithm decreases very slowly. As an illustration consider Figure 22, where two graphs for $y_n$ are plotted together: one for the conventional CUSUM algorithm and another one for our tweaked version – in both cases the simulation setup is the same, i.e. first wave of the attack is in the interval from 50.3 till 61.2 seconds and the second wave from 72.3 till 80.2 seconds of simulation scenario. We see that our tweaked CUSUM algorithm identifies the end of the attack more accurately.

![Figure 22. $y_n$ vs. time. Threshold $N = 0.1$, $\beta = 0.1$. Observation time interval is 2 seconds. Graphs for tweaked algorithm (dashed line) and conventional algorithm (solid line) are presented.](image)

To summarize, it can be noted that the choice of $\beta$ influences our detection sensitivity – the larger $\beta$ we have (the bigger offset in negatives of a sequence $z_n$) – the less likely a value will appear in $z_n$ that is going to be more than zero. In its turn, it is less likely that the test statistic will accumulate enough to merit the attack flag
being set “up”. The larger the threshold $N$, the more time delay before detecting an attack. Set threshold too high, and an attack with sufficiently low intensity will not be detected at all.

5. Conclusion

As a result of my work on the thesis, the recursive non-parametric CUSUM algorithm for detection of DDoS attacks was successfully implemented in ns-2 simulator. The analysis of its operation confirms its efficiency and robustness. The use of the developed code allows us to test numerous distributed denial of service attack scenarios in regard to the ability of the CUSUM algorithm to detect them. A possibility to use network simulation instead of relying on real-life network traces should be beneficial for further testing and development of detection techniques. At the same time it is necessary to note, that CUSUM algorithm which uses the number of new source network addresses as its statistic, though very effective and computationally simple, is still vulnerable to DDoS attacks. It is especially true when an attacker is aware of the presence of CUSUM detection mechanism at the target’s network. For example, an attacker can direct his agents to visit target’s host some time before a real attack. As these low-intensity visits usually will look like legitimate requests, the source addresses of agents will be entered into address database, i.e. considered legitimate. So as the real attack commences, the detection engine will not recognize attackers as being ones. To provide defense against this kind of a hack, we can check whether or not any source address has abnormally high traffic flow, as DDoS attacks rely not only on high number of agents but also on the high number of messages generated by every agent.

The possible future goal in continuation of the work done in my view would be to provide a working code optimized for the use as either an integral part of ns-2 distribution package or as third-party contributed module/code. In this regard we could try to implement CUSUM algorithm not only to DropTail queuing but also to
other kinds of queuing mechanisms, available in ns-2. Also some kind of algorithm optimization is necessary. For example, to use the hash table for keeping record on the number of new addresses instead of using simple arrays and thus to optimize algorithm’s performance in database lookup, speeding up its execution.

Finally, having shown that CUSUM algorithm is quite efficient for detecting DDoS attacks in simulated networking environment, the real proof of its practical validity is going to be its implementation in the real situation. For example, as Lappeenranta University of Technology possesses a sophisticated networking environment, it would be very interesting to deploy CUSUM detection mechanism at some of its leaf routers, which connect university’s network to the rest of the Internet. Certainly kernel network stack of some of the Linux based machines can be modified to include this detection engine. This would allow us to test algorithm’s performance and sensitivity under real-life conditions.
References


32. Hopper, D.I. *Denial of service hackers take on new targets.* [cited 15 November 2007]; Available from: 

33. *IBM Proventia Network Intrusion Prevention System (IPS).* [cited 1 December 2007 ]; Available from: 
   [http://iss.net/products/Proventia_Network_Intrusion_Prevention_System/product_main_page.html].


45. NS by Example - Network Components. [cited 2007 14 December]; Available from: http://nile.wpi.edu/NS/.
Appendix A

Below follows source code of the header file drop-tail-ddos.h describing the CUSUM algorithm implementation in C++.

/* -*- C++; c-basic-offset: 8; tab-width: 8; indent-tabs-mode: t -*- */
/*
 * Copyright (c) 1994 Regents of the University of California.
 * All rights reserved.
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 * must display the following acknowledgement:
 * This product includes software developed by the Computer Systems
 * Engineering Group at Lawrence Berkeley Laboratory.
 * Neither the name of the University nor of the Laboratory may be used
 * to endorse or promote products derived from this software without
 * specific prior written permission.
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
 * INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER
 * IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */

#ifndef ns_drop_tail_h
#define ns_drop_tail_h

#include <string.h>
#include "queue.h"
#include "config.h"

A bounded, drop-tail queue
class DropTailDDoS : public Queue {
public:
    DropTailDDoS() {
        q_ = new PacketQueue;
pq_ = q_;  
bind_bool("drop_front_", &drop_front_);
bind_bool("summarystats_", &summarystats);
bind_bool("queue_in_bytes_", &qib_);  // boolean: q in bytes?
bind("mean_pktsize_", &mean_pktsize_);
     // _RENAMED("drop-front_", "drop_front_");
    }
~DropTailDDoS() {
    delete q_;
    }
}

protected:
    /*struct address_database
    {
        nsaddr_t node_address;
double arrival_time;
    }*/;
} /**<
 //static address_database database;
 static nsaddr_t database_a[100];
 static double database_t[100];
 static nsaddr_t database_a_last[100];
 static double database_t_last[100];
 static nsaddr_t database_filtered[100];
 static double database_filtered_t[100];
 static int attackDetected;
 static double attackThreshold;
 static double statistic;
 static int nodeColor;
 void databaseUpdate(nsaddr_t address_, double time_);
 void database_last_Print();
 void database_current_Print();
 void databaseCheck();
 void databaseFilteredAdd(nsaddr_t address, double time);
 void databaseLastAdd();
 void reset();
 void changeColor(int color);
 int command(int argc, const char*const* argv);
 void enqueue(Packet*);
 Packet* dequeue();
 PacketQueue *q_;  /* underlying FIFO queue */
 int drop_front_;  /* drop-from-front (rather than from tail) */
 int summarystats;
 //void print_summarystats();
 int qib_;  /* bool: queue measured in bytes */
 int mean_pktsize_;  /* configured mean packet size in bytes */
};

#endif
Appendix B

Below follows the source code of the file drop-tail-ddos.cc which is the implementation of CUSUM algorithm written in C++ on the basis of DropTail queuing class.

/* -*- Mode:C++; c-basic-offset:8; tab-width:8; indent-tabs-mode:.-*-*- */
/*
* Copyright (c) 1994 Regents of the University of California.
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the above copyright
* notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
* notice, this list of conditions and the following disclaimer in the
* documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
* must display the following acknowledgement:
* This product includes software developed by the Computer Systems
* Engineering Group at Lawrence Berkeley Laboratory.
* 4. Neither the name of the University nor of the Laboratory may be used
* to endorse or promote products derived from this software without
* specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ''AS
* IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
* PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS
* BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
* SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
* INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
* CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
* ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
* OF
* SUCH DAMAGE.
*/
#ifndef lint
static const char rcsid[] =
"@(#) $Header: /nfs/jade/vint/CVSROOT/ns-2/queue/drop-tail.cc,v 1.15 2003/01/16
19:02:54 sfloyd Exp $ (LBL)";
#endif

#include "drop-tail-ddos.h"

nsaddr_t DropTailDDoS::database_a[100] = {0,0};
double DropTailDDoS::database_t[100] = {0,0};
nsaddr_t DropTailDDoS::database_a_last[100] = {0,0};
double DropTailDDoS::database_t_last[100] = {0,0};
nsaddr_t DropTailDDoS::database_filtered[100] = {0,0};
double DropTailDDoS::database_filtered_t[100] = {0,0};
int DropTailDDoS::attackDetected = 0;
double DropTailDDoS::attackThreshold = 0.3;
double DropTailDDoS::statistic = 0;
int DropTailDDoS::nodeColor = 0;
static class DropTailDDoSClass : public TclClass {
public:
    DropTailDDoSClass() : TclClass("Queue/DropTailDDoS") {};
    TclObject* create(int, const char*const*) {
        return (new DropTailDDoS);
    }
}

class_drop_tail_ddos;

void DropTailDDoS::reset()
{
    Queue::reset();
}

int
DropTailDDoS::command(int argc, const char*const* argv) {
    if (argc==2) {
        if (strcmp(argv[1], "databaseCheck") == 0) {
            databaseCheck();
            return (TCL_OK);
        }
    }
    if (argc == 3) {
        if (!strcmp(argv[1], "packetqueue-attach")) {
            delete q_;
            if (!((q_ = (PacketQueue*) TclObject::lookup(argv[2])))
                return (TCL_ERROR);
            else {
                pq_ = q_;
                return (TCL_OK);
            }
        }
    }
    return Queue::command(argc, argv);
}
/**
 * drop-tail
 */

void DropTailDDoS::enqueue(Packet* p)
{
    int iii, count = 0;
    if (summarystats) {
        Queue::updateStats(qib_?q_->byteLength():q_->length());
    }

    int qlimBytes = qlim_* meanPktsize_;

    hdr_ip *ipheader = hdr_ip::access(p);
    nsaddr_t node_number = ipheader->saddr();
    double current_time = NOW;
    //printf("packet arrived from node %d at time %f\n", node_number, current_time);
    databaseUpdate(node_number, current_time);
    //databasePrint();
    //Find out if the source address of this particular incoming packet is one of those
    //new source addresses. Done by comparing the address with a database of new
    //We do this checkup only when the node is under DDoS attack as indicated by the
    //attackDetected, which should hold true, i.e.to be equal to 1.
    if (attackDetected == 1)
    {
        for(iii=0; iii<=99;++iii)
        {
            if (database_filtered[iii] == node_number)
            {
                count = 1; break;
            }
        }
    }
    //If there is a match with one of the database entries, we just drop the packet.
    if (count == 1) { drop(p); return; }
    //Otherwise the queueing operation continues.
    else {
        if (!qib_ && (q_->length() + 1) >= qlim_ ||
            (qib_ && (q_->byteLength() + hdr_cmn::access(p)->size()) >= qlimBytes))
        {
            // if the queue would overflow if we added this packet...
            if (drop_front_) { /* remove from head of queue */
                q_->enqueue(p);
                Packet *pp = q_->deque();
                drop(pp);
            } else {
                drop(p);
            }
        } else {
            q_->enqueue(p);
        }
    }
}

Packet* DropTailDDoS::dequeue()
{
    if (summarystats && &Scheduler::instance() != NULL) {
        Queue::updateStats(qib_?q_->byteLength():q_->length());
    }
    return q_->deque();
}
void DropTailDDoS::print_summary_stats()
{
    double now = Scheduler::instance().clock();
    printf("True average queue: %5.3f", true_ave_);
    if (qib_)
        printf(" (in bytes)\n");
    printf(" time: %5.3f\n", total_time_);
}

void DropTailDDoS::database_update(nsaddr_t address_, double time_)
{
    int i;
    for(i=0;i<=99;++i)
    {
        if(database_a[i] == address_)
            database_t[i] = time_; return;
        if(database_a[i] == 0)
        {
            database_a[i] = address_;
            database_t[i] = time_; return;
        }
    }
}

void DropTailDDoS::database_check()
{
    int l, ll, ii, ee, new_addresses = 0, all_addresses = 0;
    double present_time = NOW;
    double ratio;
    double mean = 0.1;
    //Zeroing the new addresses database for further processing - adding new addresses
    for(ee=0;ee<=99;++ee) {database_filtered[ee] = 0;database_filtered_t[ee] = 0;}
    //printf("Value of attackDetected is %d\n", attackDetected);
    //Collecting the statistic during first 10 seconds of simulation
    //By default we assume that during that time there is no denial of service attacks
    //So every collected network address is legit and as such is put into database which size is 100
    if (present_time <= 10.0)
    {
        printf("Time is %f\n", present_time);
        printf("Database updated\n");
        for(l=0;l<=99;++l) {database_a_last[l] = database_a[l];database_t_last[l] = database_t[l];}
        //Assigning the static variable attackDetected false value (0 in int)
        //Somewhat redundant but for the sake of clarity let's do it
        attackDetected = 0;
        return;
    }
    //If current time is more than 10 seconds than we operate on the assumption that there could be
    //a denial of service attack, i.e. the traffic will exhibit an influx of fresh network addresses
    //We have to compare the address databases(current/updated one with the database from the last observation point),
//finding the number of fresh addresses, then normalize this number by the overall number of
//addresses and
//transform this value into negatives to allow for consecutive CUSUM summing. This value
//should be recorded into
//another static variable to keep track of it.
//if (attackDetected == 0)
printf("Comparing two databases: current and previous\n");
//printf("Last database: \n");
//database_last_Print();
//printf("Current database: \n");
//database_current_Print();
for(l=0;l<=99;++l)
{ //printf("l = \%d\n",l);
for(ll=0;ll<=99;++ll)
{ //printf("Inner cycle. ll = \%d\n",ll);
//database_current_Print();
//printf("database_a_last[%d] vs. database_a[%d] ------- \%d vs.\%d","ll,
database_a_last[l],database_a[l]);
if(database_a[l] == 0) break;
if(database_a[l] == database_a_last[ll]) break;
if(ll==99)
{
//If we did no find a match for the address in current database with any of the addresses in
//previous database, then the only possible conclusion is that it is a new address, so
//we output the corresponding message "Found... " and enter database[l] address into
database of newly discovered
//addresses by calling databaseFilteredAdd function
new_addresses = new_addresses + 1;
printf("Found \%d new addresses so far\n",new_addresses);
databaseFilteredAdd(database_a[l],database_t[l]);
}
}
//Comparison complete. New addresses are in the separate database_filtered array for later
use in filtering engine
for(l=0;l<=99;++l)
{
//Find the number of addresses discovered at previous moment of time
if(database_a_last[l] != 0) all_addresses = all_addresses + 1;
}
printf("Output of the filtered database - new addresses only\n");
int bb;
for (bb=0;bb<=99;++bb)
{
if(database_filtered[bb] != 0)
{
printf("row \%d -> Source node \%d\n",bb,database_filtered[bb]);
}
}
if((all_addresses == 0)&&(new_addresses == 0))
{printf("Attempt to divide by zero. Error \n"); return;
all_addresses = all_addresses + new_addresses;
ratio = (double)(new_addresses)/(double)(all_addresses) - mean;
printf("After comparison %d new addresses are found at time
%fn", new_addresses, present_time);
printf("Overall number of addresses is %d\n", all_addresses);
printf("Statistic is (threshold is 0.1) %f\n", statistic + ratio);
//databasePrint();
if ((statistic + ratio)>0) statistic = statistic + ratio;
else statistic = 0;
if (statistic > attackThreshold) attackDetected = 1;
else attackDetected = 0;
// If there are no new addresses during a period of observation, we equal our statistic variable
// to the attack threshold value to give us some leeway for the next time interval.
// We assume that the absence of new addresses during observation time was an indication
that
// denial of service attack stopped for real and that during next interval our statistic will
naturally fall
// below attack threshold and thus attack flag will be down. But if due to the burst nature
// there was a lag in the attack traffic and the attack is actually under way, then we presume
that attack
// traffic will still keep at next observation period our statistic variable high enough
// to merit attack flag still raised up.
if ((new_addresses == 0) && (attackDetected == 1)) statistic = attackThreshold;
printf("Statistic is %f\n", statistic);
//databasePrint();
if (attackDetected == 0)
{
  // Add new addresses into previous (in time) database - update it
  databaseLastAdd();
  printf("nodeColor is %d\n", nodeColor);
  changeColor(attackDetected);
  return;
}
printf("Denial of service attack is detected. Network addresses database update is
suspended. Filtering engine is activated.\n");
printf("attackDetected flag is %d\n", attackDetected);
printf("nodeColor is %d\n", nodeColor);
changeColor(attackDetected);
//printf("Updated database\n");
//database_current_Print();
}

void DropTailDDoS::database_current_Print()
{
  int j;
  for (j=0; j<=99; ++j)
  {
  
  
  

if(database_a[j] != 0)
{
  //printf("row %d -> Source node %d at time %f\n",j,database_a[j],database_t[j]);
}
}

void DropTailDDoS::database_last_Print()
{int j;
for (j=0;j<=99;++j)
{
  if(database_a_last[j] != 0)
  {
    //printf("row %d -> Source node %d at time %f\n",j,database_a_last[j],database_t_last[j]);
  }
}
}

void DropTailDDoS::databaseFilteredAdd(nsaddr_t address,double tiime)
{int i;
for(i=0;i<=99;++i)
{
  if(database_filtered[i] == address) {return;}
  if(database_filtered[i] == 0)
  {
    database_filtered[i] = address;
    database_filtered_t[i] = tiime;
    return;
  }
}
}

void DropTailDDoS::databaseLastAdd()
{int kkk,nnn;
for(kkk=0;kkk<=99;++kkk)
{
  if(database_filtered[kkk] == 0) break;
  for(nnn=0;nnn<=99;++nnn)
  {
    if(database_a_last[nnn] == 0)
    {
      database_a_last[nnn] = database_filtered[kkk];
      database_t_last[nnn] = database_filtered_t[kkk];
      break;
    }
  }
}
}

void DropTailDDoS::changeColor(int color)
{Tcl& tcl = Tcl::instance();
if ((nodeColor == 0) && (color == 0)) return;
if((nodeColor == 0) && (color == 1))
{
    double current_time_ = NOW;
    tcl.evalf("[Simulator instance] puts-nam-traceall \{n -t%.9f -a 0 -s 0 -S COLOR -c red -o black \}",current_time_);
    printf("Color changed to red\n");
    nodeColor = 1;
    return;
}
if((nodeColor == 1) && (color == 0))
{
    double current_time_ = NOW;
    tcl.evalf("[Simulator instance] puts-nam-traceall \{n -t%.9f -a 0 -s 0 -S COLOR -c black -o red\}",current_time_);
    printf("Color changed to black\n");
    nodeColor = 0;
    return;
}
if((nodeColor == 1) && (color == 1)) return;