# Detectability of Low-Rate HTTP Server DoS Attacks using Spectral Analysis

Joel Brynielsson*† and Rishie Sharma*
*KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden
†FOI Swedish Defence Research Agency, SE-164 90 Stockholm, Sweden
Email: {joel, rishie}@kth.se

***Abstract*—Denial-of-Service (DoS) attacks pose a threat to any service provider on the internet. While traditional DoS flooding attacks require the attacker to control at least as much resources as the service provider in order to be effective, so-called low-rate DoS attacks can exploit weaknesses in careless design to effectively deny a service using minimal amounts of network traffic.**

**This paper investigates one such weakness found within version 2.2 of the popular Apache HTTP Server software. The weakness concerns how the server handles the persistent connection feature in HTTP 1.1. An attack simulator exploiting this weakness has been developed and shown to be effective. The attack was then studied with spectral analysis for the purpose of examining how well the attack could be detected.**

**Similar to other papers on spectral analysis of low-rate DoS attacks, the results show that disproportionate amounts of energy in the lower frequencies can be detected when the attack is present. However, by randomizing the attack pattern, an attacker can efficiently reduce this disproportion to a degree where it might be impossible to correctly identify an attack in a real world scenario.**

***Index Terms*—Low-rate DoS attack; attack simulator; Apache HTTP Server; attack detection; spectral analysis.**

## I. Introduction

In 2003 Kuzmanovic and Knightly [1] published an article on a weakness found in the Transmission Control Protocol (TCP) congestion control mechanism which opened up for a different, more intelligent, type of DoS attack. TCP is the transport protocol used by most internet traffic and the weakness thus pose a serious threat to many online services. By appropriately timing burst attacks (spikes) the attacker can use much less traffic than a traditional brute force DoS attack while still achieving considerable DoS. This type of attack has also been called Reduction-of-Quality (RoQ) attack, shrew attack, and pulsing attack. Because of the low amount of traffic used compared to a traditional brute force attack, the low-rate attack is supposedly harder to detect. In [1] the authors argue that while the effects of the attack can be mitigated by using for example randomization, the attack cannot be completely defended against without significantly sacrificing system performance in the event of legitimate congestion. There are other attacks of this nature that exploit timing mechanisms to achieve high amounts of damage with relatively low amounts of traffic. An example given in [2] is a video service where a video can be viewed by users. An attacker could abuse this service by knowing the length of the video and periodically performing requests at intervals of the same length as the video. Another attack described in [2] exploits the HTTP protocol and targets a feature in HTTP 1.1 called persistent connection or keep-alive. By abusing the persistent connection feature the attack can considerably degrade an HTTP server's ability to serve legitimate clients while using minimal amounts of traffic. The attack is called LoRDAS in the article and is described as a general attack where the HTTP persistent connection attack is an example of this general attack.

A traditional DoS attack detection scheme might use volume based detection where a disproportionate amount of traffic would indicate a DoS flood attack. That type of detection is not applicable to low-rate DoS attacks, however, since they use minimal amounts of traffic. Another traditional approach is to use signature based detection on incoming packets but that approach will only work if there is something in the attack packets that distinguish them from ordinary packets which is not necessarily the case for DoS attacks. A slightly more sophisticated approach is the one used in [3] where they gain positive results by using maximum entropy estimation to detect anomalies in network traffic. In [4], Barford et al. use wavelet analysis, which is a type of spectral analysis, to effectively detect anomalies in traffic data. Similarly, Chen et al. study the low-rate DoS attack on TCP using spectral analysis in [5]. In their research they found that network traffic data containing the low-rate attack has more energy in the lower frequency band as compared to legitimate traffic.

This paper aims to investigate how well the HTTP persistent connection DoS attack can be detected using spectral analysis. Inspiration is taken from the work done by Chen and Hwang [5], [6], [7], where they study the low-rate DoS attack on TCP using spectral analysis. This paper, however, instead aims to study the low-rate DoS attack on HTTP as described in [2] in order to see how well that attack can be detected using spectral analysis. As part of the project, a simulator for the HTTP attack has been built and shown to be effective. The simulator only aims to target the popular Apache HTTP Server (referred to as simply Apache throughout the paper) as to not overcomplicate the simulator. The Apache version to be attacked is version 2.2. Apache's access log data from when under attack by the simulator is analyzed using spectral analysis for the purpose of finding distinguishing features.

## II. BACKGROUND

Section II-A describes the HTTP persistent connection feature whilst Section II-B explains how Apache handles connections. Section II-C is dedicated to explaining the LoRDAS attack, and Section II-D finally explains the basic concepts of spectral analysis that are used in this paper.

### A. HTTP Persistent Connection

When a web page is loaded in a web browser there are often several images and other items embedded in the web page. If a new TCP connection is opened for every item that is required from the web server, overhead would be introduced because of the communication required by TCP when establishing and closing a connection. By instead reusing a single TCP connection for several requests, resources can be saved: fewer packets are sent thus reducing network congestion, latency on subsequent requests is reduced since they do not require handshaking, and by using a single connection, HTTP requests and responses can be pipelined. Pipelining refers to sending multiple requests without waiting for responses in-between [8].

Because of the performance benefits of reusing a TCP connection, the persistent connection feature was introduced in HTTP 1.1 as default behavior. The feature is sometimes referred to as keep-alive. Persistent connection leaves a TCP connection open for a certain amount of seconds after a request to a HTTP server has been made to allow for further requests. The specification does not define how long the connection should be left open, and leaves it up to the client and server to close the connection when they see fit.

In Apache the persistent connection feature is controlled by the parameters KeepAliveTimeout and MaxKeepAliveRequests [9], [10]. The KeepAliveTimeout parameter controls how many seconds a connection is kept open after a request. In Apache 2.0 the default timeout was 15 seconds but in version 2.2 and 2.4 it is as short as 5 seconds. It is not clear from the Apache documentation whether the timeout is reset upon subsequent requests, but it turns out that it is. That is, if the KeepAliveTimeout is 5 seconds and a request is made at 12:00:00 followed by a subsequent request using the same connection at 12:00:02, the connection will be open for a total of 7 seconds. The MaxKeepAliveRequests parameter controls how many times a connection can be reused for sending new requests before it is closed. This parameter defaults to 100. If MaxKeepAliveRequests is set to 0, an infinite number of requests can be made using the same connection.

### B. Apache Connection Handling

The way Apache handles connections depends on the Multi-Processing Module (MPM) used. In Apache 2.2 there are 7 MPMs available [11]. For the NetWare operating system the mpm_netware MPM is available, for the OS/2 operating system the mpmt_os2 MPM is available, for BeOS the beos MPM is available and for the Windows operating system the mpm_winnt MPM is available. Having mentioned that, this report will continue to only focus on the three MPMs available for Unix-related operating systems which are prefork, worker and event. The event MPM was considered to be experimental in Apache 2.2 [12] and prefork was the default MPM. In version 2.4, event is considered to be stable [13] and is the default MPM used for any system that supports both threads and thread-safe polling. In practice that means all modern operating systems [14].

*1) Prefork:* The prefork MPM does not use threads and instead uses a separate child process for every connection [15]. This MPM is useful when the system does not support threads, or non-thread-safe libraries are being used. For prefork the maximum number of child processes that will be launched is regulated by the ServerLimit parameter whose default value is 256. The maximum number of connections that Apache can service simultaneously is controlled by the MaxClients (called MaxWorkerRequests in Apache 2.4 [16]) parameter which also has the default value of 256. In practice ServerLimit becomes the upper limit for MaxClients for prefork, since prefork only uses processes to serve connections.

*2) Worker:* The worker MPM uses both processes and threads to service connections [17]. The number of processes is controlled by the ServerLimit parameter and its default value is 16. Each process can have a number of threads that serve connections and the maximum number of threads a process can have is limited by the ThreadsPerChild parameter. ThreadsPerChild has a default value of 25 which means that the maximum number of the total number of threads is $16 \cdot 25 = 400$ per default. As for the prefork MPM the MaxClients parameter controls the maximum number of simultaneous connections but is limited by the product of ServerLimit and ThreadsPerChild instead of only ServerLimit.

*3) Event:* The event MPM is similar to the worker MPM except that instead of keeping a separate thread for each connection that is being kept open because of the persistent connection feature, all connections that are in that keep-alive state are handled by a thread dedicated to handling such connections along with other idle connections. This frees up threads that otherwise would be waiting for subsequent requests that may or may not come. If a new connection attempt is made and all workers are busy, the event MPM will close connections in keep-alive state to free up a position even if the timeout has not expired. This has dire consequences for the feasibility of the LoRDAS attack.

*4) When the Server is Full:* When Apache cannot accept more requests because there are no more processes or threads available, requests will be queued. The amount of requests that are queued are limited by the ListenBacklog directive [18] which defaults to 511. However, this value is often limited to a lower value by the operating system. The server that has been used to run Apache in the study presented herein was running a Linux 3 kernel which limited the number of incomplete sockets in the queue to 256 and limited the number of completed sockets to 128. More information about how Linux handles the backlog is available in the man page for the listen system call[1].

[1] http://linux.die.net/man/2/listen

### C. LoRDAS

The so-called low-rate DoS attack against application servers, LoRDAS, is described in [2]. The attack is described as a general attack against any server using the following model. A server can serve several requests simultaneously and may be composed of a single machine or several machines connected to a load balancer. Each machine can serve a limited number of users simultaneously. In Apache 2.2, using the prefork MPM, that limit would correspond to the MaxClients directive. Each machine has its own service queue where requests that cannot be handled immediately are queued, and this queue has a limited length. In Apache this corresponds to the ListenBackLog directive.

When the service queue is full in all machines, new connections cannot be handled and will be discarded. This results in a Denial-of-Service which is the primary goal of the LoRDAS attack. LoRDAS aims to fill the service queue with malicious requests so that legitimate requests are discarded. To achieve this, a regular DoS flood is sufficient. However, LoRDAS tries to achieve the same result using as little network traffic as possible. By predicting the points in time at which a position in the service queue becomes available, LoRDAS can time its attacks accordingly and thereby greatly reduce its overall network traffic use.

To be able to predict these instants, LoRDAS needs to exploit some vulnerability in the server. For the case of Apache using the prefork or worker MPM, this vulnerability can be the persistent connection feature. When employing the event MPM there is no obvious way to predict when a position becomes available in the service queue since a position will be made available if there are idle connections in keep-alive state connected to the server.

### D. Spectral Analysis

Spectral analysis concerns the study of spectra of “the distribution of power over frequency of a time series” [19].

*1) Signals and Domains:* Spectral analysis may be considered to be a subfield of signal processing, and shares its terminology and concepts with this field. The most basic concept is the idea of a signal. A signal is “a function that conveys information about the behavior or attributes of some phenomenon” [20] and includes examples such as sound signals, electrical signals, optical signals, and electromagnetic signals. Any measurable quantity such as one’s weight, the temperature in the fridge, the number of birds on a windowsill per day, etc., can be turned into a signal.

Two concepts important for understanding signal processing and spectral analysis are the time domain and the frequency domain. When performing a scientific experiment one might measure the value of some sort of signal, for example the temperature, at specific times and record the measured values along with the time at which each measurement was made. Doing this one obtains a set of data points where each data point contains a temperature and a time. This set of data points is said to be in the time domain since the value of our signal is in relation to a specific time.

In the time domain one can easily see when a signal took upon what values. In the frequency domain, on the other hand, a signal is represented as a sum of many periodic functions. By representing a function in this way, one can easily see periodic behavior such as oscillations or fluctuations. If one for example measures the temperature outside every 10 minutes one would probably be able to see a daily fluctuation of the temperature in the frequency domain. Now, one would be able to see that daily fluctuation by simply graphing the values of the signal in the time domain as well but it is not always that easy to see the fluctuations. An example where it would be very hard to see the fluctuations by graphing the signal in the time domain would be the sound signal of an orchestra playing a symphony. In this case the fluctuations are vibrations of air molecules caused by many different instruments that form many different sound frequencies which combined create musical chords. To be able to answer questions such as “in which key is the song being played?” or “is the orchestra playing in tune?,” one would have to transform the sound signal to the frequency domain first.

*2) Transforms:* To transform a signal from the time domain to the frequency domain and back, one uses transforms. Mathematically speaking, a transform is simply a function but the word transform is used instead of the word function for certain applications such as when rotating the points of a triangle or, as in our case, when transforming a signal. Since we will be working with a finite set of points describing a signal in the time domain and we want to transform it to the frequency domain, a transform we can use is the Discrete Fourier Transform (DFT). Another transform we can use is the Discrete Wavelet Transform (DWT). There is an important difference between the two transforms. The DFT gives an output that is totally independent of time, meaning that there is no way to tell when certain frequencies occurred in the input. One can only tell which frequencies occurred. If we need to be able to tell when certain frequencies occurred we can either divide the input into small (often overlapping) segments and apply the DFT on each segment (the smaller segments the better time precision but worse frequency precision), or use a DWT. In this study we will only be interested in if frequencies occur rather than when frequencies occur which is why we will be using the DFT.

The DFT takes a sequence of complex numbers as input and outputs a sequence of complex numbers. The length of the output sequence is equal to the length of the input sequence. The following equation gives the $k$:th number in the output sequence of the DFT where $N$ is the length of the input (and output) sequence, $x$ is the input sequence and $X$ is the output sequence:

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-i(2\pi k \frac{n}{N})}. \quad (1)$$

Computing the DFT using the above definition requires $O(N^2)$ operations but there is an algorithm called the Fast Fourier Transform (FFT) which computes the DFT using $O(N \log N)$ operations.

Conceptually speaking the DFT checks which frequencies best match the input and gives a high value output for frequencies that match well and a low value output for frequencies that do not match well. Mathematically speaking the $e^{-i(2\pi k \frac{n}{N})}$ part of the formula is simply a circle in the complex plane with radius 1 and as $n$ increases we rotate around this circle. If $x(n)$ does not match up well with the speed we are rotating around this circle (controlled by the $k$ parameter), then the total sum will average up to around zero because the points $x(n) \cdot e^{-i(2\pi k \frac{n}{N})}$ in the complex plane become evenly spread around the origin like in the right-hand side of Fig. 1.

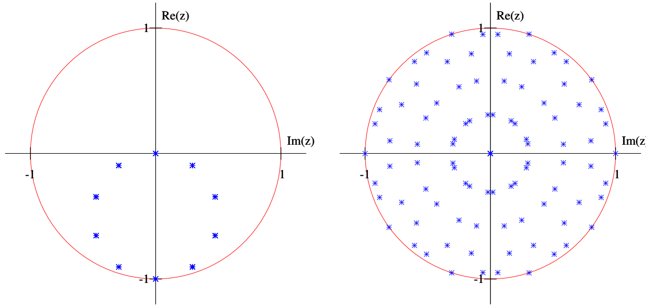


Fig. 1. Here we use a series of 100 points, $x(n) = \sin(2\pi n/20)$ where $n \in \{0, 1, .., 99\}$, and plot the complex points $x(n) \cdot e^{-i(2\pi kn/100)}$ for $k = 5$ to the left and for $k = 6$ to the right. Since $2\pi n/20 = 2\pi kn/100 \Rightarrow k = 5$, we have that choosing $k = 5$ will make the two periodic functions revolve with the same speed and thus skewing the resulting complex points towards one direction. Choosing $k = 6$ will instead cause the two periodic functions to not match up at all, and the resulting complex points will be evenly spread around the origin.

However, if $x(n)$ fluctuates with a speed equal to the speed we are rotating around the complex circle with, the points $x(n) \cdot e^{-i(2\pi k \frac{n}{N})}$ become skewed away from the origin like in the left-hand side of Fig. 1. This is because all the high points and low points of the fluctuation in $x(n)$ will be multiplied with the same angle during each rotation around the complex circle. Instead of the points becoming evenly spread around the origin, the rotation around the complex circle and the fluctuation in $x(n)$ adds up to move the center of the points away from the origin. This gives us a total sum whose magnitude (distance from the origin) is large.

*3) Sample Rates, the Nyquist Frequency, and Aliasing:* There are some limitations and side effects that stem from the fact that we represent our signals as sequences of data points, often referred to as samples of a signal, instead of continuous functions. This is not to say that we could actually measure a continuous function representation of a real world signal. Continuous functions are only relevant in theory; in the real world we have to deal with finite sequences of measurements. The most obvious limitation is that we cannot know what goes on between samples. The sampling frequency is the amount of samples we have per second and the sampling frequency directly limits what frequencies we will be able to detect with the DFT.

The Nyquist frequency, named after the Swedish engineer Harry Nyquist (1889–1976), is defined to be half of the sampling frequency. The reason it is important is that as long as the input signal does not contain any frequencies above the Nyquist frequency the original signal can be perfectly reconstructed using the sampled signal. This fact is formulated in the Nyquist-Shannon sampling theorem [21]. What this means for us is that when transforming our signal with the DFT we will not be able to detect frequencies higher than the Nyquist frequency and if there are frequencies higher than the Nyquist frequency in the input, they will be aliased by lower frequencies.

Aliasing is when high frequencies (larger than the Nyquist frequency) look like lower frequencies to us because our sample rate is too low to capture those frequencies. This phenomenon can be seen when filming some rotating object (car wheels, helicopter rotor blades, etc.) and the rotation appears to be rotating slower, be stationary, or even go in the reverse direction to the actual rotation. This occurs because the sample rate of the camera might be 24 FPS (frames per second) or 24 Hz which gives a Nyquist frequency of 12 Hz and a helicopter tail rotor might spin with a speed of 1500 RPM (Revolutions Per Minute) which gives a frequency of 25 Hz. 25 Hz is greater than 12 Hz so the 25 Hz frequency will be aliased with another frequency. To be able to see what frequency 25 Hz will be aliased with in this case, one can imagine the frequency, $f$, as a clock hand revolving around a clock face at the speed of $f$ and the sample rate, $s$, is how often one looks at the clock. If $s$ and $f$ are equal, then each time one looks at the clock the clock hand will appear to not have moved at all because it will have moved exactly one rotation. In fact, as long as $f$ is a multiple of $s$ then the clock hand will have moved a whole number of rotations and it will look like the hand has not moved at all. When $s$ is 24 and $f$ is 25 as in our example then when one looks at the clock face the clock hand will have rotated slightly more than one rotation which ends up looking like the clock hand has only moved slightly. To be precise, we are looking at the clock every 24th of a second and $f$ is rotating 25 rotations per second which gives that when we look at the clock, the clock hand will have moved $\frac{25}{24} = 1 + \frac{1}{24}$ rotations which will end up looking to us like it has moved only $\frac{1}{24}$ rotations and this will be indistinguishable from if $f$ was 1 Hz.

To avoid aliasing one can remove frequencies higher than the Nyquist frequency with a filter before sampling a signal. This is referred to as bandlimiting.

## III. Methodology

This section explains the setup used to test the attack in Section III-A and continues to describe how the attack simulator works in Section III-B. The section ends by describing how the spectral analysis was performed in Section III-C.

### *A. Laboratory Setup*

To perform a LoRDAS attack in practice one would have to control a large botnet. It would not be possible to use IP address spoofing to fool the target that an attack is coming from multiple directions while it in fact is coming from only

one attacker. This is not possible because a web server uses HTTP which is built on top of TCP, and TCP uses a three-way handshake to establish a connection. What that means is that if a TCP packet with a faked source IP address is sent to the target with the intention of opening a TCP connection, the target will send a corresponding confirmation TCP packet with the very important randomly chosen sequence number back to the faked IP address and the attacker would never receive it.

To avoid setting up a large botnet in order to be able to perform experiments, a special setup was used. The target web server was configured to route all packets through the attacker which means that the attacker receives everything the target sends out. In this way the attacker can establish faked TCP connections with only one computer using a multitude of IP addresses. If one wanted to perform an attack in reality then this setup is of course not possible since it requires root access to the target, and if one has root access to the target then performing a DoS attack would be an unnecessary complication.

The Apache version used was a modified version 2.2.25 (details about the modification can be found in Section III-C). All the default settings were used, which means that the prefork MPM was used, MaxClients was set to 256, ServerLimit was set to 256 and ListenBacklog was set to 511. The document that was requested a countless number of times was a standard HTML web page: `<html><body><h1>It works!</h1></body></html>`.

The server was running Arch Linux with Linux kernel version 3.11.4. The server was connected to a router with a 100 Mbit/s Ethernet connection. The attacker was connected to the same router with a 100 Mbit/s Ethernet connection. The attacker was running Ubuntu with Linux kernel version 3.8.0. The Python version used to run the attack program was Python 3.3.1. The router used was a Linksys E900 running the 1.0.04 firmware.

### B. LoRDAS Simulator

The simulator was implemented as a Python 3 program. The program uses raw sockets to send IP packets without automatically prepended IP headers so that a forged IP header with a fake source IP address can be prepended by the program.

There is one thread in the program that monitors all network packets coming in and out of the computer (requires root access) and filters out any packets that are not TCP packets with a source address of our target. From the remaining packets the program checks if there is a queue created by a bot thread for the destination IP address found in the packet and, if so, places the packet in that queue. The queues are thread-safe objects which are part of the standard Python 3 library.

Apart from the packet listening thread there are the bot threads. Each bot runs in a separate thread. The bot threads are initialized with a specific source address and they then create a queue for this source address so that packets targeted at this source address are picked up by the packet listener and placed in that queue. Using this queue the bot tries to establish a TCP connection with the target using the TCP three-way handshake. After the handshake has been completed a HTTP GET request is sent to the target and if all goes well the bot then sleeps for a certain amount of seconds. The amount of time spent waiting is crucial for how much traffic will be used and for the detectability of the attack. Whatever value is used, it must be less than the KeepAliveTimeout of the target.

After the short sleep the bot performs another HTTP GET request to refresh the KeepAliveTimeout timer on the server. This behavior is repeated until MaxKeepAliveRequests is reached and the bot then basically restarts itself trying to open a new TCP connection. One might wonder how long a bot waits before for example resending a SYN packet if no SYN+ACK packet is received or if no response is received to a HTTP GET request. A timeout of 5 seconds was used for simplicity, but it should optimally be set to conform with the behavior of major web browsers so that a bot's behavior cannot be distinguished from legitimate users in this sense.

### C. Spectral Analysis of Access Logs

Apache logs all incoming requests in an access log. This log contains a time stamp for each request but the time stamp does not contain any higher precision than whole seconds by default. If we were to use a time stamp with only second precision then that would mean that our sample rate would be 1 Hz. A sample rate of 1 Hz would lead to a Nyquist frequency of 0.5 Hz which would mean that we could not capture any periodic behavior with intervals shorter than 2 seconds.

Apache 2.4 has the option to change the format of the time stamp to include milliseconds and even microseconds since the Epoch (often January 1, 1970) [22]. This option is not present in Apache 2.2, however, which led to that the Apache 2.2 mod_log_config source code was patched so that tests could be performed with Apache 2.2 as well. The change was made in the mod_log_config.c source code file in Apache version 2.2.25.

To convert the access log to a signal that we can transform to the frequency domain with the DFT, a simple script was used. The script goes through every line of the access log, extracts the time for each line and outputs a number for each time unit corresponding to the number of requests received during that time unit. When requests were coming extremely fast, the requests in the access log were in disorder so the requests had to be sorted before being converted to a signal.

To illustrate how a signal was created, the following log (normal time stamps are used here because they are easier to understand):

```
18:00:00
18:00:00
18:00:02
18:00:04
18:00:05
```

would be converted to the following signal with a time unit of 1 second:

2 0 1 0 1 1

Note that each number in the output of the conversion corresponds to a time unit (in this example, a second).

The time unit chosen for the experiments performed was 1 ms, which was chosen without much thought. A time unit of 1 ms corresponds to a sample rate of 1000 Hz which leads to a Nyquist frequency of 500 Hz. Decreasing the time unit even more would help us capture periodic behavior on scales larger than 500 times per second which are time scales that seem irrelevant in network traffic. In [6] the authors similarly used a sample rate of 1000 Hz.

After converting the access log to a digital signal, the absolute value of the FFT of the signal was plotted against a sequence of evenly spaced numbers from 0 to the sample rate (1000 Hz) that had the same length as the input signal. The absolute value of the FFT of the signal was placed on the y-axis and the numbers between 0 and 1000 were placed on the x-axis. Since we cannot detect any frequencies above the Nyquist frequency (500 Hz), the plot was cut in half so that the x-axis ran from 0 Hz to 500 Hz.

To measure the disparity between lower and higher frequencies the following calculation was performed for $d = 50$, 25, 10 and 5:

$$q = \frac{\sum\limits_{0<x<d} f(x)}{\sum\limits_{\substack{500-d<x \\ x<500}} f(x)}, \qquad (2)$$

where $f$ is the function that given a frequency outputs the energy for that frequency. The equation serves to calculate the ratio between the energy in a band of lower frequencies and the energy in a band of higher frequencies. The lengths of the bands are controlled by $d$. If there is no disparity between the bands and the energy levels are equal, $q$ becomes 1. If, for example, there is twice as much energy in the lower band compared to the higher band, $q$ becomes 2.

## IV. Results

The purpose of this section is to determine whether our methods are capable of distinguishing attacks from random data using many different attack configurations. For each attack we use the $q$-values that are calculated with Equation 2 to determine whether the attack is distinguishable from random data. Since random data has $q$-values that are more or less equal to 1, we are trying to find a certain attack configuration that produces $q$-values close to 1. Such an attack would be indistinguishable from random data and thus undetectable using our methods.

The results are presented as graphs with frequency on the $x$-axis and the slightly ambiguous unit "energy" on the $y$-axis. Energy refers to the absolute value of the output of the DFT which means that the more energy a frequency has, the stronger that frequency is in the signal.

Along with every graph a moving average with a window of 100 samples is also plotted so that trends can more easily be seen. The moving average is aligned so that the first value of the moving average graph is the average of the first 100 points and the moving average graph is 100 points shorter than the main graph.

For each attack the $q$-values are also calculated according to Equation 2 with $d = 50$, 25, 10 and 5. Each attack is performed 10 times to provide an idea of the variation in the results. The mean and standard deviation of the 10 tries are presented in a table.

### A. Fixed Wait Times

Using fixed wait times means that each bot waits for a fixed period of time between each request it makes to the server. In Fig. 2, using a fixed wait time of 4.5 s., we can see that the energy levels in the lower frequencies are much higher than in the higher frequencies. For this wait time, the $q$-values in Table I show that the energy levels in the lower frequencies are twice as high as in the higher frequencies. This means that there is much frequency periodic behavior in the signal. The low frequency periodic behavior we see is created by the attack bots.

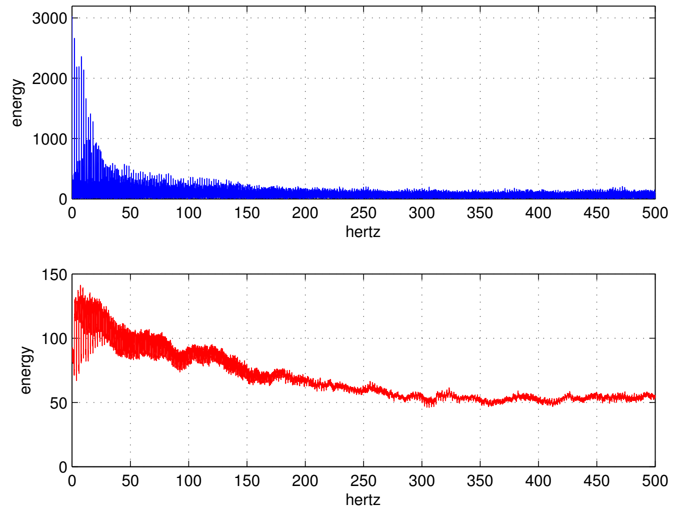


Fig. 2. The upper subplot shows the spectral content of an attack of length 60 s. using 400 bots where each bot had a wait time of 4.5 s. between requests. The lower subplot shows the moving average using 100 samples of the upper subplot.

As can also be seen in Table I, reducing the wait times results in greater energy levels in the lower-most frequencies. This does not mean that shorter wait times produce lower frequencies, but simply that there are more data points, i.e., a shorter wait time produces more data points per second.

### B. Random Wait Times

In Fig. 3 we see the energy distribution over frequency for a simulated attack using bots with random wait times evenly distributed between 0 and 4.5 seconds. Compared to the fixed time graphs we see much more noise in Fig. 3. There is still more energy in the lower frequencies which can be confirmed in Table I, but the difference is much smaller now when we use random wait times as opposed to when we used fixed wait times.

TABLE I

THE TABLE SHOWS THE VALUE OF $q$ ACCORDING TO EQUATION 2 FOR DIFFERENT COMBINATIONS OF THE BANDWIDTH VARIABLE $d$ AND THE BOT WAIT TIMES. THE $q$-VALUE IS A MEASUREMENT OF THE AMOUNT OF ENERGY IN THE LOWER FREQUENCIES AS COMPARED TO THE HIGHER FREQUENCIES. EACH $q$-VALUE REPRESENTS TEN DIFFERENT ATTACKS, WHICH IS PRESENTED USING MEAN VALUE AND STANDARD DEVIATION.

| Wait time [s] | $d$ [Hz] | $q$ |
|---|---|---|
| 4.5 | 50 | $2.0759 \pm 0.1199$ |
| | 25 | $2.1722 \pm 0.1632$ |
| | 10 | $2.1572 \pm 0.1393$ |
| | 5 | $2.0512 \pm 0.1210$ |
| 2 | 50 | $1.8119 \pm 0.0779$ |
| | 25 | $2.1658 \pm 0.0742$ |
| | 10 | $2.8377 \pm 0.1544$ |
| | 5 | $3.3872 \pm 0.2704$ |
| 0.5 | 50 | $1.4935 \pm 0.0768$ |
| | 25 | $1.9300 \pm 0.1088$ |
| | 10 | $2.8643 \pm 0.1686$ |
| | 5 | $3.7461 \pm 0.2324$ |
| $\mathcal{U}(0, 4.5)$ | 50 | $1.3539 \pm 0.0532$ |
| | 25 | $1.3694 \pm 0.0492$ |
| | 10 | $1.3889 \pm 0.0317$ |
| | 5 | $1.4199 \pm 0.0567$ |
| $\mathcal{U}(0, 2)$ | 50 | $1.2358 \pm 0.0527$ |
| | 25 | $1.2731 \pm 0.0503$ |
| | 10 | $1.3557 \pm 0.0590$ |
| | 5 | $1.4742 \pm 0.0707$ |
| $\mathcal{U}(2, 4.5)$ | 50 | $1.4635 \pm 0.0361$ |
| | 25 | $1.4760 \pm 0.0470$ |
| | 10 | $1.4834 \pm 0.0427$ |
| | 5 | $1.5023 \pm 0.0673$ |

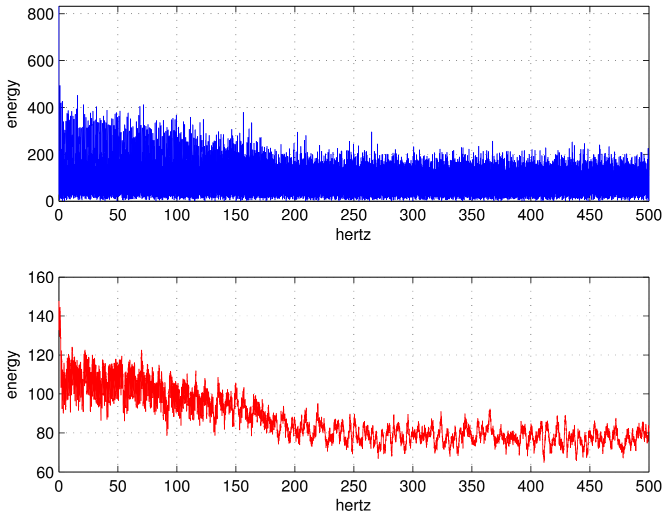


Fig. 3. The upper subplot shows the spectral content of an attack of length 60 s. using 400 bots where each bot had a random wait time between 0 and 4.5 s. between requests. The lower subplot shows the moving average using 100 samples of the upper subplot.

## V. Attack Phases and its Relation to Detectability

During the experiments it could be seen that increasing the detection time length produced more stable results in general, i.e., with less standard deviation when calculating $q$ using Equation 2. Since we restart the attack for each test this might signify that the beginning of an attack is more detectable than the continuation of it. Looking closer at a 300 s. attack whose $q$-values are $1.0571, 1.0565, 1.0668, 1.0856$ we get the following $q$-values for the first circa 30 s.: $1.4701, 1.4755, 1.6132, 1.7972$; the following $q$-values for the last circa 30 s.: $0.9910, 0.9650, 0.9482, 0.8849$; and the following $q$-values for circa 30 s. in the middle of the attack: $0.9968, 0.9930, 0.9745, 1.0306$. This seems to confirm that the beginning of the attack is more detectable than the ongoing attack.

The beginning of the attack being more detectable is most likely related to how the attack bots are started. One could imagine that the bots somehow are not evenly spread out to begin with and then after some time of using random wait times the bots become evenly spread out and harder to detect. Changing how the bots were started did not seem to have any effect, though. The following startup techniques were investigated: starting the bots as fast as possible, using a small random wait time between the startup of each bot, and trying to start the bots evenly spread out along a Poisson distribution. None of these changes changed the fact that the first 10–15 seconds of the attack resulted in higher $q$-values than the rest of the time. The explanation had to be that there is something fundamental behind the startup of the attack that is more easily detected.

When looking closer at what happens when the bots are started, we can see that most of the bots never receive a SYN+ACK packet for their first SYN packet to initiate a TCP connection. This is understandable since the server cannot handle 400 new TCP connections at once. What happens then is that the bots who did not manage to open a TCP connection will retry after 5 seconds (because that is how they were programmed, see Section III-B). Even after the second round there will be bots who still did not manage to open a TCP connection, and they will wait for another 5 seconds. In this way there will be spikes every 5th second during the startup of the attack, and this is what lies behind the increased $q$-values at the beginning of the attack. If we wanted to remove the detectability of the initiation of the attack, the attack would have to employ a slow-start as to not flood the server while starting the bots.

It is important to point out that the above does not mean that the attacks presented in the previous section are undetectable. The attack startup is still present in all attacks, and the attacks still show different detectability using different bot wait times. This means that apart from the startup phase, the wait times must also affect the detectability.

## VI. Conclusions

While the attack described in this paper will be successful against a default configured Apache 2.2 server, it has no effect on a default configured Apache 2.4 server. This is due to the prefork MPM being used by default in version 2.2 while the event MPM is used by default in version 2.4. The event MPM

keeps all connections that are being kept alive because of the persistent connection feature in a separate thread dedicated for such connections. When the server needs a slot for a new connection and there are no more slots available it will close one of the idle connections that are being kept alive. In this way, even if the connection that was closed was a legitimate connection, only a small overhead will be introduced the next time a document is required from the server because of the need to open a new connection. On the other hand, the attack described herein is completely thwarted because a bot cannot hold a connection for the purpose of blocking new connections, since new connections are always prioritized over idle connections.

At the time of writing (spring 2015), Apache 2.4 has been available for more than three years and is considered stable by the Apache HTTP Server Project. However, major GNU/Linux distributions such as Ubuntu and Arch Linux do not provide the 2.4 version by default and when Apache is installed they provide the 2.2 version. Furthermore, many sites, such as the web site of our own university, still use Apache 2.2.

When it comes to detection of the attack using spectral analysis, this is possible as long as the attacker uses fixed wait times or floods the server when initiating the attack. If the attacker is clever, however, the attack can be made undetectable for spectral analysis through using random wait times and a sufficiently slow start phase. This does not necessarily mean that a spectral analysis detection scheme is useless, though. As shown [4], spectral analysis can be used to detect anomalies in everyday traffic data. In [4], the analysis is made on IP packets and SNMP data, but it might suffice to perform spectral analysis on a higher abstraction level similar to herein. As part of an anomaly detection system, the methods developed in this paper, such as calculating $q$-values using Equation 2, might prove useful to detect certain types of anomalies.

To detect the attack presented in this paper, one could also consider each connection by itself and try to evaluate if a connection is a legitimate connection or an attacker bot. One could, e.g., look at which documents are being requested and at what times, and compare legitimate user behavior with bot behavior. If a connection is suspected to be a bot connection, that connection can be prohibited from keeping its connection alive and forced to initiate a new connection. Another approach could be to check the amount of idle connections versus the amount of active connections and temporarily disable the persistent connection feature if the ratio is too high.

## References


[1] A. Kuzmanovic and E. W. Knightly, "Low-rate TCP-targeted denial of service attacks (the shrew vs. the mice and elephants)," in *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM'03. New York, NY: ACM, Aug. 2003, pp. 75–86. [Online]. Available: http://dx.doi.org/10.1145/863955.863966

[2] G. Maciá-Fernández, J. E. Díaz-Verdejo, P. García-Teodoro, and F. de Toro-Negro, "LoRDAS: A low-rate DoS attack against application servers," in *Critical Information Infrastructures Security: Second International Workshop, CRITIS 2007*, ser. Lecture Notes in Computer Science, J. Lopez and B. M. Hämmerli, Eds. Berlin/Heidelberg, Germany: Springer-Verlag, 2008, vol. 5141, pp. 197–209. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-89173-4_17

[3] Y. Gu, A. McCallum, and D. Towsley, "Detecting anomalies in network traffic using maximum entropy estimation," in *Proceedings of the 2005 Internet Measurement Conference*, ser. IMC'05. Berkeley, CA: USENIX Association, 2005, pp. 345–350.

[4] P. Barford, J. Kline, D. Plonka, and A. Ron, "A signal analysis of network traffic anomalies," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet Measurment*, ser. IMW'02. New York, NY: ACM, 2002, pp. 71–82. [Online]. Available: http://dx.doi.org/10.1145/637201.637210

[5] Y. Chen, K. Hwang, and Y.-K. Kwok, "Filtering of shrew DDoS attacks in frequency domain," in *Proceedings of the 2005 IEEE Conference on Local Computer Networks*, ser. LCN'05. Piscataway, NJ: IEEE, Nov. 2005, pp. 786–793. [Online]. Available: http://dx.doi.org/10.1109/LCN.2005.70

[6] Y. Chen and K. Hwang, "Collaborative detection and filtering of shrew DDoS attacks using spectral analysis," *Journal of Parallel and Distributed Computing*, vol. 66, no. 9, pp. 1137–1151, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2006.04.007

[7] ——, "Spectral analysis of TCP flows for defense against Reduction-of-Quality attacks," in *Proceedings of the 2007 IEEE International Conference on Communications*, ser. ICC'07. Piscataway, NJ: IEEE, Jun. 2007, pp. 1203–1210. [Online]. Available: http://dx.doi.org/10.1109/ICC.2007.204

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1," Internet RFC 2616, Jun. 1999. [Online]. Available: http://dx.doi.org/10.17487/RFC2616

[9] The Apache Software Foundation. Apache HTTP Server Version 2.2 Documentation: KeepAliveTimeout Directive. [Online]. Available: http://httpd.apache.org/docs/2.2/mod/core.html#keepalivetimeout

[10] ——. Apache HTTP Server Version 2.2 Documentation: MaxKeepAliveRequests Directive. [Online]. Available: http://httpd.apache.org/docs/2.2/mod/core.html#maxkeepaliverequests

[11] ——. Apache HTTP Server Version 2.2 Documentation: Multi-Processing Modules (MPMs). [Online]. Available: http://httpd.apache.org/docs/2.2/mpm.html

[12] ——. Apache HTTP Server Version 2.2 Documentation: Apache MPM event. [Online]. Available: http://httpd.apache.org/docs/2.2/mod/event.html

[13] ——. Apache HTTP Server Version 2.4 Documentation: Apache MPM event. [Online]. Available: http://httpd.apache.org/docs/2.4/mod/event.html

[14] ——. Apache HTTP Server Version 2.4 Documentation: Multi-Processing Modules (MPMs). [Online]. Available: http://httpd.apache.org/docs/2.4/mpm.html

[15] ——. Apache HTTP Server Version 2.2 Documentation: Apache MPM prefork. [Online]. Available: http://httpd.apache.org/docs/2.2/mod/prefork.html

[16] ——. Apache HTTP Server Version 2.4 Documentation: MaxRequestWorkers Directive. [Online]. Available: http://httpd.apache.org/docs/2.4/mod/mpm_common.html#maxrequestworkers

[17] ——. Apache HTTP Server Version 2.2 Documentation: Apache MPM worker. [Online]. Available: http://httpd.apache.org/docs/2.2/mod/worker.html

[18] ——. Apache HTTP Server Version 2.2 Documentation: ListenBackLog Directive. [Online]. Available: http://httpd.apache.org/docs/2.2/mod/mpm_common.html#listenbacklog

[19] P. Stoica and R. L. Moses, *Spectral Analysis of Signals*. Upper Saddle River, NJ: Prentice Hall, Inc., 2005.

[20] R. Priemer, *Introductory Signal Processing*, ser. Advanced Series in Electrical and Computer Engineering. Singapore: World Scientific, 1991.

[21] C. E. Shannon, "Communication in the presence of noise," *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, Jan. 1949. [Online]. Available: http://dx.doi.org/10.1109/jrproc.1949.232969

[22] The Apache Software Foundation. Apache HTTP Server Version 2.4 Documentation: Apache Module mod_log_config. [Online]. Available: http://httpd.apache.org/docs/2.4/mod/mod_log_config.html