Anomaly-Based, Intrusion Detection Visualizations

By Kevin Phillip Galloway

B.S., University of Alaska, Fairbanks, 2008.

PROJECT

Submitted in partial fulfillment of the requirements

for the degree of Master of Computer Science

in the Graduate College of the

University of Alaska Fairbanks 2010

Table of Contents

Abstract

Due to the large amounts of data and ever increasing number of new attacks, methods to quickly investigate and analyze this data is incredibly important. In this paper we seek to demonstrate and describe a visualization that will help researchers detect anomalies in data. A short case study is included as well, discussing how this visualization may be used to quickly judge how useful an algorithm is in finding anomalies.

Acknowledgements

First, I would like to thank my committee, Dr. Hay, Dr. Lawlor, and Dr. Chappell, for their help on this project, and for igniting my love for this field. I would especially like to thank Dr. Hay who's been a good sport when dealing with the abuse his graduate students have heaped upon him, and for introducing me to the computer security field.

I would like to thank other graduate students: Brandon Marken, Brian Paden, Walker Wheeler, Wesley McGrew, and John Styers, for listening to me ramble on about my project, for helping me shape the visualization, and for listening in general.

Finally I would like to thank my fiancee, Michelle Vieira, for sticking with me as I went through all this, and helping me keep focus, when my attention strayed from the project.

1. Introduction

1.1 Visualization Introduction

Due to the rapidly increasing amount of data in computer security, it is becoming imperative that tools be created to quickly analyze and evaluate data. Visualization is one method to examine large amounts of data, before further analysis, to get a high-level view, before deeper analysis happens. Most visualization tools focus on easily quantifiable variables, such as those found in a packet capture. For example, a packet capture can consist of packet length, port number, and other numerical data such as these. These are typically numeric, and easily parsed, as each number is separate, and individually grouped. However, there are a wide variety of tools that return data in simple strings, such as high interaction honeypots and keyloggers, yet few tools that specifically parse or collate data from text, with most work in this field simply listing the number of times that a word was listed.

1.2 Problem Statement

1.2.1 Sequential Order of Events

Many visualizations focus on analyzing trends over time, but do not focus on the sequential ordering of events. The standard line graph, for example, will show an overview of its data source, and the frequency of each category in its data source. However, for certain data sources, the order and sequence of events is instrumental to understanding what exactly is happening with the data. In command line data, for example, knowing the sequence of commands might shed some information on what an attacker was trying to do, or if the attack was automated.

1.2.2 Textual Data

Textual data, as a whole, can be difficult to collate. There are numerous words in the English language, for example, that are related in meaning, but not in letters. Take searching words that mean “thin” in a body of text. Most simple parsers would simply keep track of how many times the word "thin" appears in this text. However, a user might want to keep track of synonyms of thin, and group them together, so words like "slender", "thin", and "lean" all contribute to the same value.

1.3 Project Scope

In this project, we provide a possible solution for each of these problems. We create a visualization that displays each event in a dataset. We define an event as an entry in a line of data at a particular timestep. Each event is displayed sequentially, showing the order of each event, what follows and event, and what comes before an event.

In regards to dealing with the textual data, we created a flexible matching implementation, that allows users to group keywords together. Users can also assign colors and the height displacement for any event string that matches this group of keywords in the visualization.

The scope of this project is to create a system that will allow a user to easily link in a parser that separates a piece of data into three variables: a timestamp, a source, and an event string. The parser must be able to be written by a user, and easily connected to the visualizer, so that users may use any data source they wish, and so that we do not have to write a parser for every data type. The visualizer will display the events in a sequential line of points, with matched events being a user-defined color.

2. Background

2.1 Related Work

In 2004 Stefan Schlechtweg and Ragnar Bade, from the University of Magdeburg, and Silvia Miksch from the Vienna University of Technology applied a *Lifelines* approach for time-oriented data in the medical field. This technique used color and height to display critically elevated, elevated, normal, reduced, and critically reduced qualitative levels for fever temperatures [1]. While not directly related to computer security, a similar approach may be applied to security data. Many fields use visualizations to analyze their data, and certain techniques are useful in multiple fields, such as the *Lifelines* approach. Using both color and height to differentiate data helps to distinctly show the different events in a data set.

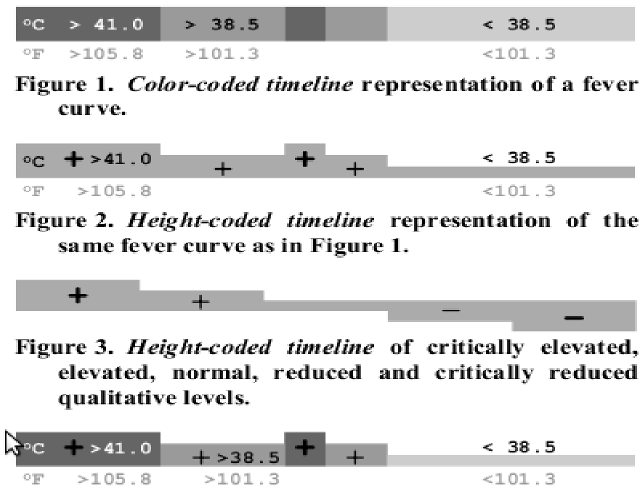


Figure 1. ***Color-coded timeline*** **representation of a fever curve.**

Figure 2. ***Height-coded timeline*** **representation of the same fever curve as in Figure 1.**

Figure 3. ***Height-coded timeline*** **of critically elevated, elevated, normal, reduced and critically reduced qualitative levels.**

Figure 1: Lifeline's Approach

H. Gunes Kayacik and A. Nur Zincir-Heywood of Dalhousie University applied Kohonen's Self-Organizing Feature Maps algorithm to the KDD-Cup 1999 data to build a visual model of known attacks. Instead of analyzing the the flow of information, that is, how the traffic moves from one machine to the other, this algorithm focuses on attack behavior. One can compare the visual representations of a known attack with potential zero-day attacks to examine if there are any similarities [2].

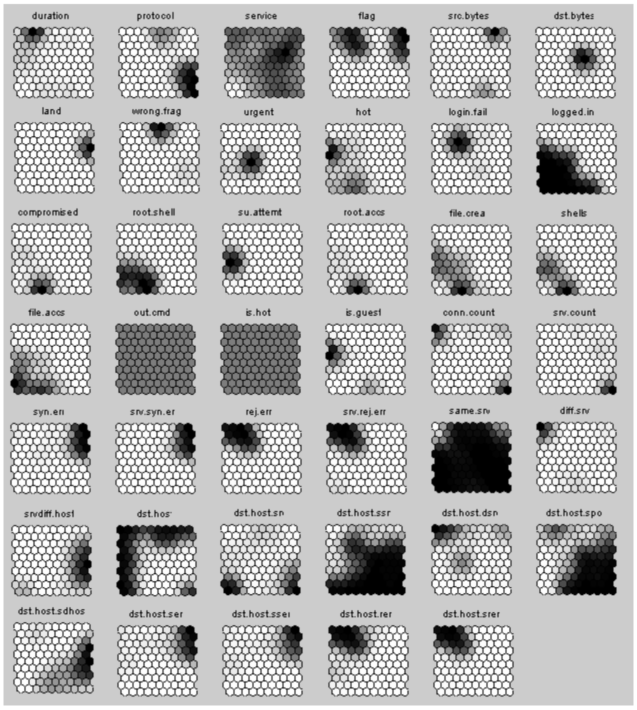


Figure 2: KDD-Cup 1999 Anomaly Visualization

KNAVE-II was created by Asaf Shabtai, Denis Klimov, Yuval Shahar, and Yuval Elovici in 2006 to visualize a group of subjects at various levels of abstractions [3]. Their tool handles both individual and multiple subjects to create multiple graphs of different data sources and types.

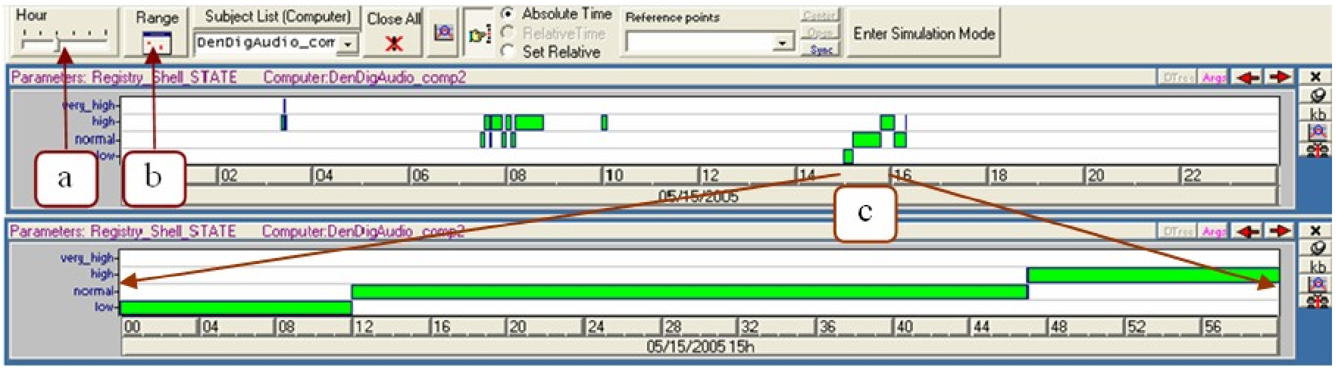
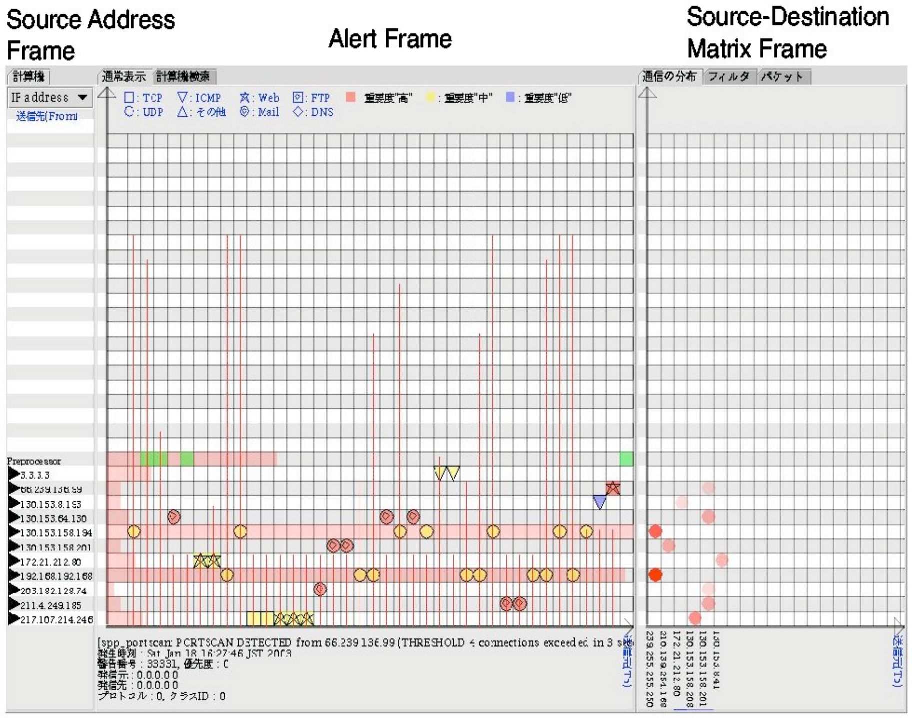


Figure 3: Demonstration of Time Granularity Controls

SnortView was created by Koiki and Ohno [5] to display data taken from the intrusion detection system, Snort. This visualization attempts to visualize data that is not simply numeric. Different alerts that Snort outputs are represented by colored symbols, and lines represent the timeline of an event.



```
[spp_portscan: PORTSCAN DETECTED from 66.239.136.99 (THRESHOLD 4 connections exceeded in 3 se
発生時刻 : Sat Jan 18 16:27:46 JST 2003
警告番号 : 33331, 優先度 : 0
発信元 : 0.0.0.0 0
発信先 : 0.0.0.0 0
プロトコル : 0, クラスID : 0
```

Figure 4: SnortView

These tools all show data in different ways, as well as different types of data, and all are useful for their specific targets. However, the wide variety of tools to display these anomalies, shows that there are no standard ways to highlight anomalies in a dataset. Furthermore, none of these tools focus on textual data, nor do they focus on how sequential events may be connected.

2.2 Visualization Terms

In visualization, there are common terms and elements that one can follow to create images that users will be able to read simply, as such we will define these terms here.

Pre-attentive visual properties: Pre-attentive visual properties [7] are the visual elements that are immediately noticed, such as coloring an element of a visualization a separate color from the others, or having a different size. These properties can include the form, color, position, and motion.

Exception emphasis: Visualizations should seek to emphasize exceptions by highlighting, in one form or another, important data. This can be done by using a separate color for exceptional events [7].

Gestalt Principles: Gestalt principles are visual characteristics that allow users to either separate, connect, or highlight data quickly in a visualization. The proximity of visual elements to one another, and how they are connected together are examples of Gestalt principles[7].

2.3 Data Sources

2.3.1 Honeypots

In the field of computer security one of the methods of detecting threats, whether they be new or old is the use of what is called a Honeypot [8]. Put simply a Honeypot is an application that simulates the processes of a physical machine so that a potential attacker or malicious user thinks that he or she is interacting with a real machine. The Honeypot then records the interaction between the malicious user and the simulated machine. Honeypots are generally broken down into two categories, high interaction Honeypots and low interaction honeypots.

High interaction Honeypots typically simulate an entire machine, such that it is essentially a machine placed onto a network or the internet to be attacked. These typically have keyloggers and other ways to record a user's keystrokes, such that a researcher can see exactly what an attacker was doing. Since these are typically full machines one can use them to launch further attacks. Setting up a full network of high interaction honeypots is challenging as each high interaction honeypot is basically a real machine.

Low interaction honeypots do not simulate full machines, but rather vulnerabilities or specific components. Due to this, a low interaction honeypot will fail when an attacker wishes to compromise the machine, it will just have vulnerabilities that are exploited. However, one may through up a simulated network of thousands of machines using low interaction honeypots as they simply masquerade as services.

Data from high interaction honeypots present a much different challenge from the data gathered from low interaction counterparts, due to the types of data that one collects. With a low interaction honeypot, it is fairly trivial to collect and normalize the data to create visualizations. Most low interaction honeypots simply keep track of packets, numeric data that is fairly trivial to categorize.

We used data collected from Sebek, a kernel module for honeypots, which stores data from the command line. Along with the command line output, it also gathers the date, host, User ID, Process ID, File Descriptor, Inode number, and the communication port.

2.3.2 PCAP Files

PCAP files are network traffic data, recorded from every packet that travels on a network. In this project, we focus specifically on the comma-separated value format exported from Wireshark by default. Wireshark is a network packet analysis tool, that can read live network packets, or recorded PCAP data files. In this format, there are six fields: Packet number, Timestamp, Source IP address, Destination IP address, network protocol, and an Information field.

Separating a PCAP CSV is trivial, especially compared to working with High Interaction Honeypot data. There are several libraries in a variety of languages to parse CSVs, and we use the default Python CSV library to separate the fields.

2.3.3 KDDCup 1999 Data

In the intrusion detection field, the data for the KDDCup 1999 [6] competition is typically used as a standard set to test new tools on. Despite the age of this dataset, it was used in a brief test to determine how useful this visualizer could be to test the output or data matching of algorithms, partly to show another possible use of this visualizer.

The KDDCup 1999 Data is generally used for anomaly-based intrusion detection systems, and as such is a combination of both dangerous packets along with normal traffic. Each packet is recorded as a comma-separated list of roughly 25 features. There are data files involved, but the only files of interest in regards to this project is unlabeled and labeled test data files.

Sebek
[2009-05-12 08:39:05 Host:192.168.13.4 UID:0 PID:4108 FD:3 INO:13021 COM:sshd]#SSH-2.0-OpenSSH_4.3
PCAP CSV
"1","0.000000","10.42.42.253","10.42.42.50","TCP","46104 > http [SYN] Seq=0 Win=5840 Len=0 MSS=1460 TSV=3299940 TSER=0 WS=6"
KDD-Cup 1999
0,udp,private,SF,105,147,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2,0.00,0.00,0.00,0.00,1.00,0.00,0.00,255,255,1.00,0.00,0.01,0.00,0.00,0.00,0.00,0.00

3. Original Design

The original design for the parser was a Python script that would format the data into a certain format, so that a C++ program could read this format, and extract the relevant pieces of data to compare with user-defined keywords. A window would contain every string of data that the parser had read, so that a user could scroll through them to determine what keywords they wished to look for, organized through a GUI. However, this proved impractical, as it was just as simple for a user to look at the raw text file themselves, as the GUI did not offer any compelling advantage.

The original design for the visualizer was something of a particle engine. Each particle would represent an event, and depending on the timestamp, would flow from a center point, moving either toward the top, bottom, left, or right part of the screen. The most recent events would be toward the center of the screen, while the older events would be toward the edges, and all of this would be animated. Animation, however, was rejected in the final build because it requires the researcher to pay constant attention to the visualization. Furthermore, without some sort of tracing system (tails tracing out the point's path, for example), it is difficult to see exactly how an event changes from one timestep to the next.

4. Experimental Design

4.1 The Parser

The visualizer is divided into two parts, called the Parser, and the Visualizer. Due to the original purpose of the visualizer, to examine data from the Sebek Honeypot, the focus of the parser is on textual data. This presents its own challenges that one may not encounter in numeric data. A string of text may be related to another string, despite having unrelated characters. A naïve algorithm might simply try to match an entire line of text with a keyword or a keyphrase. For example, a simple matching algorithm may not take a command's flags into consideration. If we use the ls command as an example, an algorithm must take into account that ls -a and ls -l are the same command but have different flags. It may be important that the flags are different, or it might not be, and this is a challenge that appears with textual data, but not with numeric.

The parser itself is a simple script that reads in data and outputs it into into a list of Event objects. These objects are specified to have three variables: a timestamp, a source, and the event string. The timestamp is simply the number attached to the ordering of the events in a user's data, while the source is where the data originated from. Both of these can be any kind of data. The timestamp, for example, can be the time the event happened, to the packet number, while the source can be a source IP or the specific machine name. The event string is left up to the user, but it is what the matching implementation seeks to match a group with.

The matching implementation allows users to define a grouping of keywords. Since command line data is very much non-standardized, allowing user-defined keywords, and more importantly user-defined groupings is important. Many tools and commands in the security field are easily grouped together, nessus and nmap for example, are used for network reconnaissance and security vulnerability assessment, and a user might want to group these tools together, to see if they are found in the command line data. This approach also allows users to place future tools or commands into a group, without having to contact the developer.

User-defined groupings are implemented through a Group object. To interact with the visualization, a Group must have a list of the keywords to match in the Event string, the color to display, if a keyword matches the Event string, and a height for the point to appear if, again, a keyword matches the Event string. A Group object reads from a grouping file, that the user creates, which is a file consisting of a list of two line entries. Each entry contains the color, in RGB format, and the height on the first line, followed by the character separated keywords. For a more thorough explanation, pleease see Appendix A: Visualizer Use, Section *Grouping Files*.

The matching implementation was written in Python, specifically to interact with the parsing portion, as well as allowing the user to include regular expressions in their grouping file, without the user having to install another library on top of the other libraries they have to install to use the visualizer. Allowing this also allows the user to pack a variety of data into their Event string in the parsing portion, and to use regular expressions to match multiple variables in an Event string.

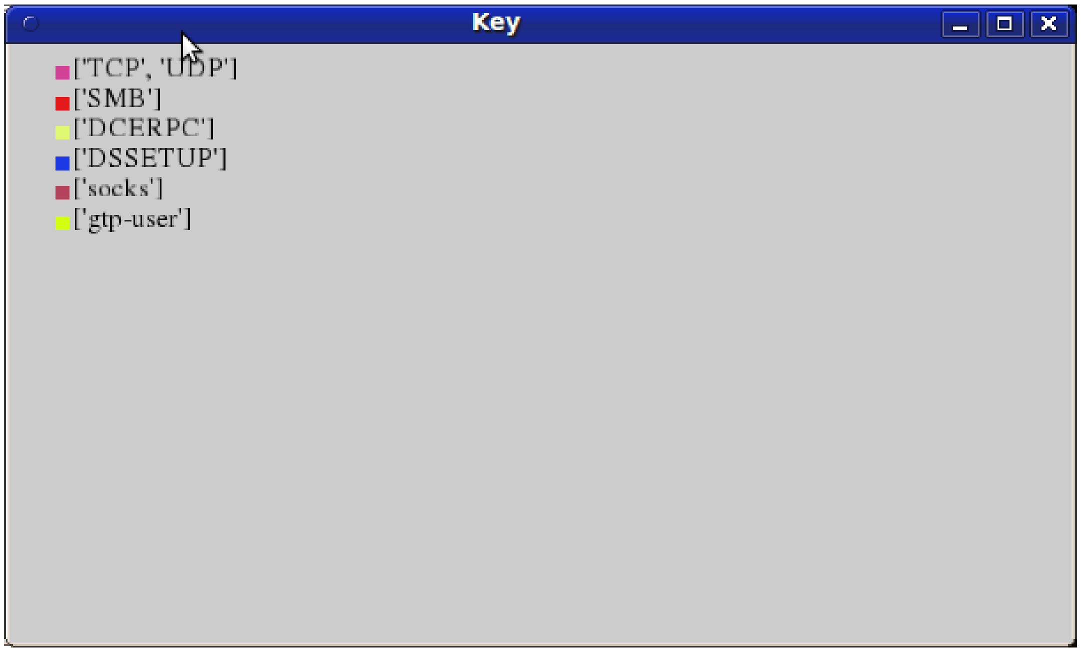
4.2 The Visualizer

The final version of the visualizer was written in Python, specifically so that it could interface with the parser in a much easier fashion. Therefore it would be less work for a user to write a parser for their own data source, and have it interact with the visualizer without dealing with a complicated intermediary file between a parsing script, of any language, and the C++ visualizer. This approach allows the flexible matching implementation to interface with the visualizer.

One requirement for the visualizer was that it would be cross-platform, easily run, and provide a visualization of adequate speed. To accomplish this, OpenGL was decided upon, yet Python has no native OpenGL support. Therefore, a library called Pyglet [9], which is a Python wrapper around OpenGL, with user interface functions included, was used.

Due to prior research, such as the *Lifelines* approach, the visualizer uses two different visualization styles to display the data from the parser. *Lifelines* used a technique that separates events by both color and height. Both styles have multiple elements in common. First, both styles display roughly 45 events, with events proceeding along the x-axis, which represents event number, moving from earlier events to later events as the points move to the right. Second, sources are displayed along the y-axis, moving upward in both visualizations. Thirdly, to accommodate the potential different sizes of each data source in a visualization, every tenth point is larger to show how the user is stepping through a set of data, and when the data stops.

The visualizer is broken into three windows, one displaying the key for the data, with the other two displaying the two separate visualization types. The key windows simply displays a colored point, followed on the right, by the list of keywords that will show up as that color.



🟪 ['TCP', 'UDP']
🟥 ['SMB']
🟨 ['DCERPC']
🟦 ['DSSETUP']
🟥 ['socks']
🟨 ['gtp-user']

Figure 5: Example of the data key

The top-left window displays the data solely based on color. Separating events by color helps give this visualization a pre-attentive property; it is easy to tell which events are different.

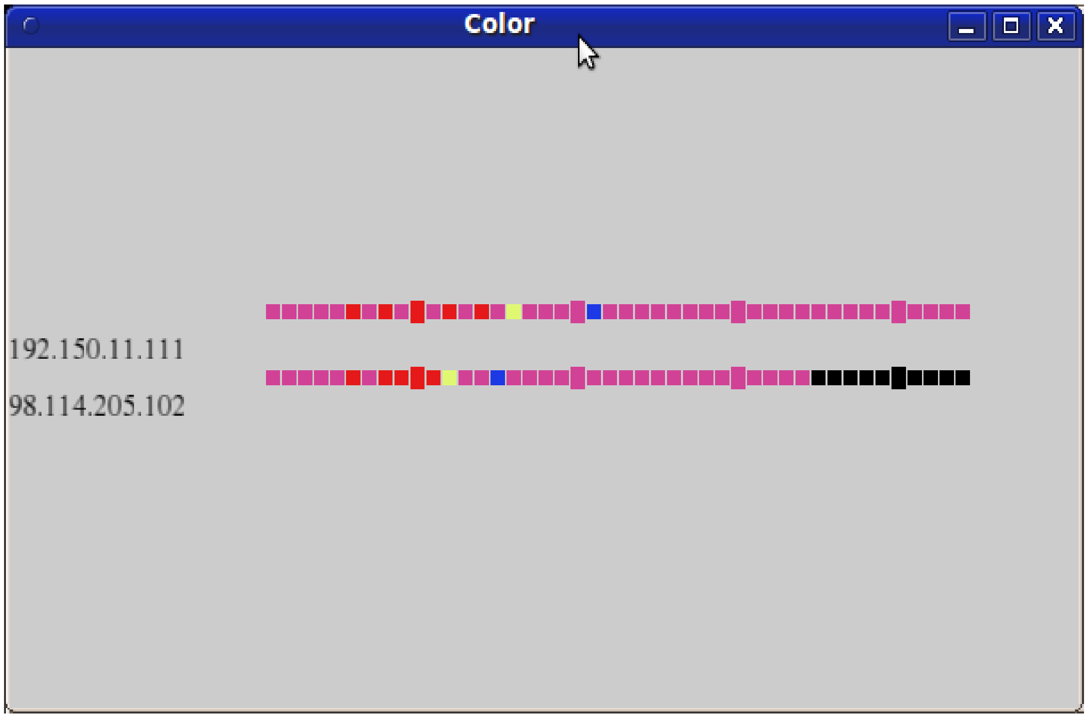


Figure 6: Color differentiated visualization

The bottom-left window displays the data differentiated by both color and height. Much like the simple color separated image, this type gives the visualization two pre-attentive properties: the different colors, and the different height, making exceptions stand out.

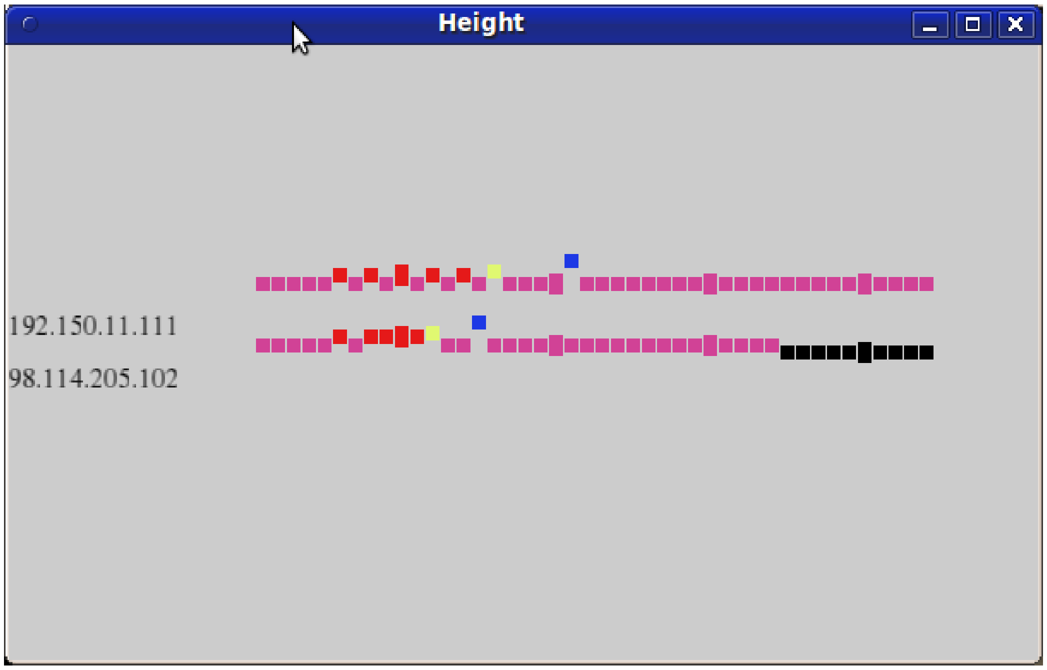


Figure 7: Color and height differentiated visualization

Both of the windows that display data can be stepped through using the arrow keys, to allow a user to look through the data at their own speed. Each visualization type also shows off the Gestalt properties, with the points positioned by source, and events in sequence placed next to one another.

4.3 KDDCup 1999 Data Experiment

Using the KDDCUP 1999 data was an attempt to discern if this visualization could be used for other purposes than just detecting anomalies. To this end, we tried using a simple matching algorithm to find a smurf attack in this dataset, and compared it with the labeled test data. A smurf attack is a kind of denial of service attack, using the ping command. To create the keywords, we wrote a simple script that would take the KDDCUP 1999 labeled data, and place the list of features, in a string, into the grouping file. Since we were also going to look through the labeled data, a final group was appended at the end, with the keyword “smurf”, to show which events were smurf attacks in the corrected data.

The list of events came from two sources. The first was the unlabeled test data, with the source “kdd_1999_unlabeled”, while the second was the labeled test data, with the source “corrected”. Both of these datasets have essentially the same data, though the labeled set has an extra field which is the attack type. These were placed into the list of events so that we could compare our naïve algorithm, with the actual results to see if this visualization could be used to show the effectiveness of an algorithm.

Our visualizer creates the following image when used with these parameters:

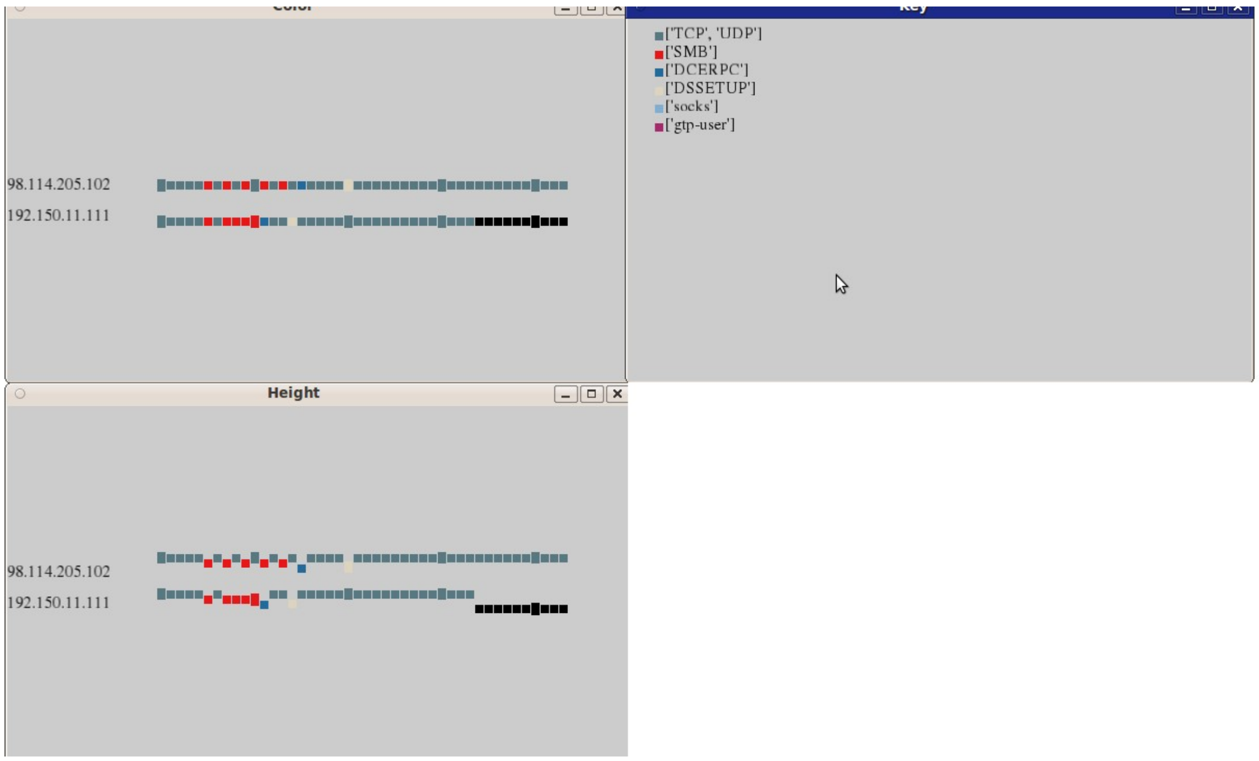


Figure 8: Full visualization example

5. Experimental Results

5.1 Sebek Results

The data collected from the Sebek honeypot was simply test data, and used for the first round of tests. While the data itself was simply random keystrokes, this data proved to be a solid demonstration of the visualizer and the parsing. While there was no data analysis for this dataset, it provided a quick demonstration of the visualizer and the parser.

For the Sebek dataset we created a parser that specifically combined the year, month, day and the current time into the timestamp, took the Host IP as its data source, and grabbed all text after the '#' mark, to use as the Event string. We created a grouping file with the following entries:

```
ping,ssh
0.9,0.1,0.1,0.05
test,nK,exit
.sbin.,PATH
```

In this example, any events that match “ping” or “ssh” would be displayed as a single random color and a single random height, “test”,”nK”, or “exit” would be displayed as a mostly reddish color, at the height of .05, and “.sbin” or “PATH” matched events would be assigned a single random color and a single random height.

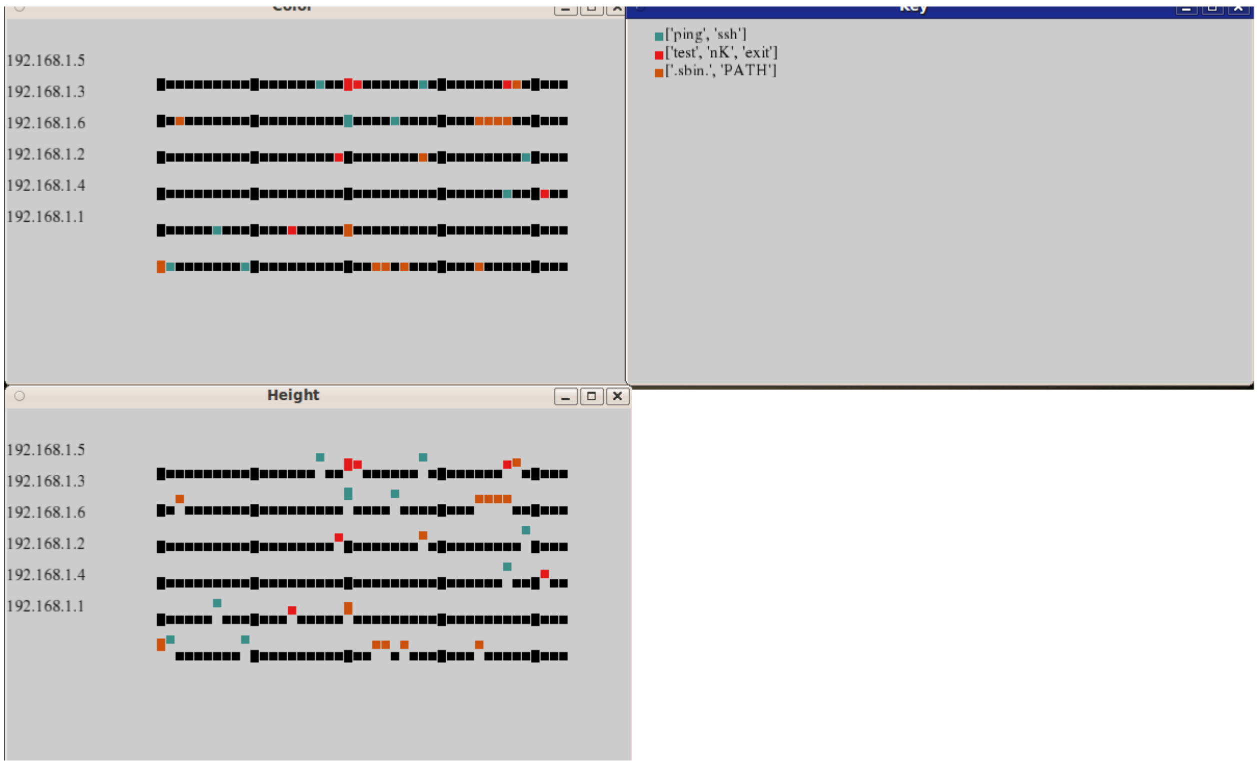


Figure 9: Example of Sebek Data Visualization

In the above example, the red points represent event strings with the keywords 'test', 'nK', and 'exit' in both windows. In the bottom left window, one can see that the red points are elevated at a certain height as well.

5.2 KDDCup 1999 Data Results

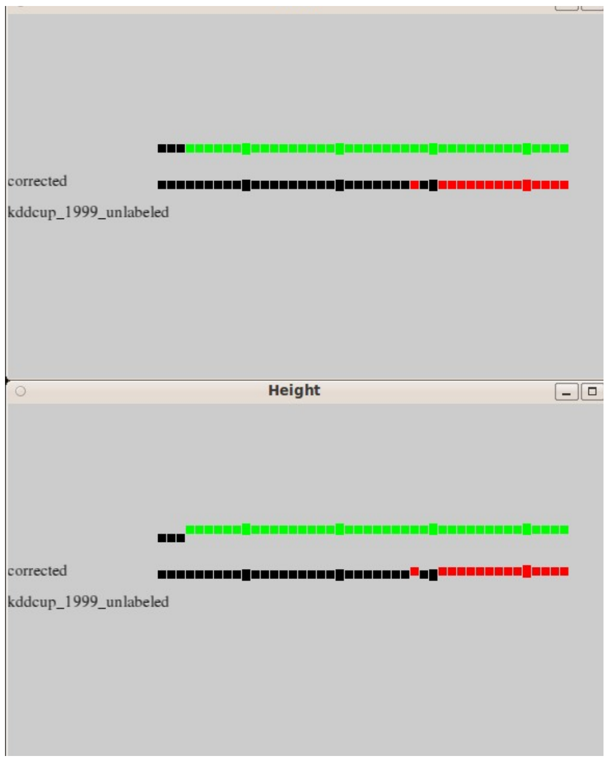


Figure 10: KDD-Cup 1999 Smurf Attack visualization

By using this visualization to investigate the usefulness of a naïve matching algorithm, to find anomalies with the KDDCup 1999 data, we saw that a naïve matching algorithm does not work. This algorithm, which simply matched every smurf attack example in the test data, missed multiple occurrences of the Smurf attack. However, we have demonstrated that this visualization could be used to analyze an anomaly detection algorithm visually.

In the above visualization, the top bar represents the correct KDD-Cup 1999, labeled data, with each green event being an actual smurf attack. The lower bar is the unlabeled test data, with which we used our naïve algorithm. Each red point is an event that our naïve algorithm matched. By comparing the corrected data, with the unlabeled, one can see how our algorithm missed many of these events.

6. Future Work

This visualizer can be expanded in multiple ways. The more trivial changes would be to the user interface. Displaying the event number, allowing more points to be shown on screen, as well as adding mouse support would be changes to increase the usefulness of this visualizer. These improvements would allow a greater degree of feedback for users, granting them the ability to get a clearer idea of where they are in a dataset.

Further research could focus on using this visualizer to examine common datasets in the computer security field. Many of the tests shown in this paper were more of a proof of concept, showing how exactly the visualizer works and looks, or were demonstrated on problems where other, non-visualization based methods would work as well. The KDD-Cup 1999 data, for example, can be examined using mathematical algorithms. Using this visualization to analyze a dataset that has not been extensively examined by other methods would show the use of this visualization.

Appendix A: Visualizer Use

Installation

The visualizer is developed in Python2.6 using the pyglet library (http://www.pyglet.org/). The pyglet library uses OpenGL for its graphical capabilities. The source files for the Visualizer are as follows:

- visualizer.py: This is the main function for the Sebek visualizer, and calls the parser function that one wishes to use.
- draw.py: This holds the drawing functions for all three windows of the visualizer.
- event.py: This holds the class for each Event
- parser.py: This is the parsing function that will read in the data, and convert it into an Event.
- group.py: This holds the class for each Group object

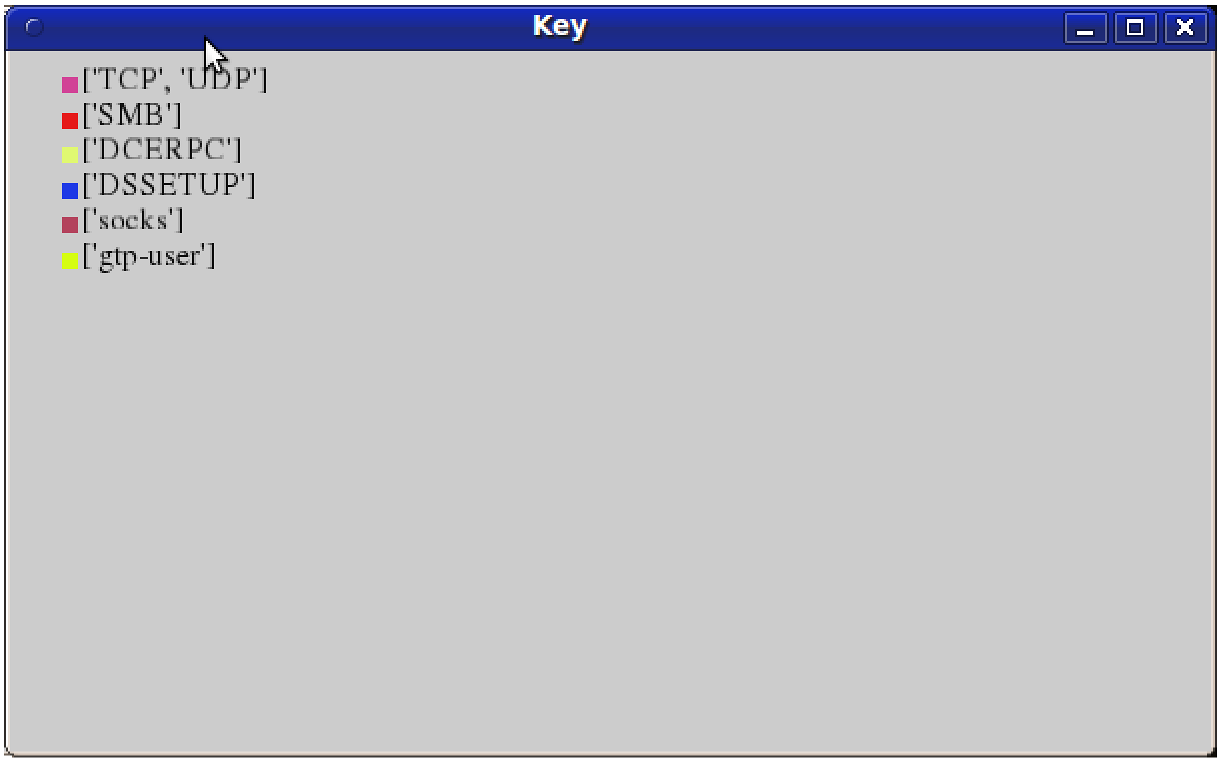
- size.py: This is a simple class that holds the sizes for each point

Once one has all the files, and the libraries installed, one can simply run the visualizer:

```
$ python visualizer.py
```

Reading the Visualization

After the visualizer is run three windows will appear, each showing data:



🟪 ['TCP', 'UDP']
🟥 ['SMB']
🟨 ['DCERPC']
🟦 ['DSSETUP']
🟥 ['socks']
🟨 ['gtp-user']

Figure 11: Color key for each group

The first window is the key for each user defined group. The colored point preceding the text in brackets shows what color each user-defined group is represented by. Each word or text surrounded by a single quote is one keyword to be searched for as a part of that group.

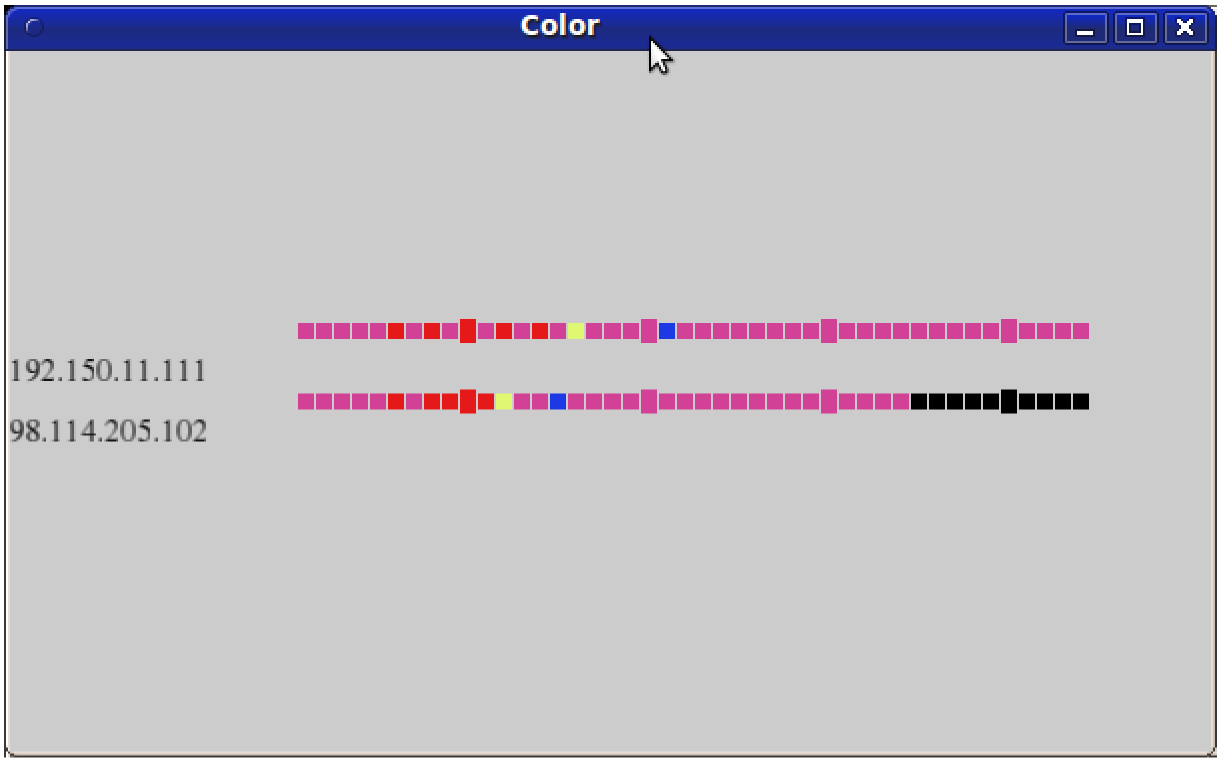


Figure 12: Figure of the color differentiated graph

This window displays a color differentiated graph: events are differentiated only by color. Unmatched events appear as black, and the IP addresses of each host, each with a list of data, are displayed on the left. Finally, a larger point it placed at every tenth interval.

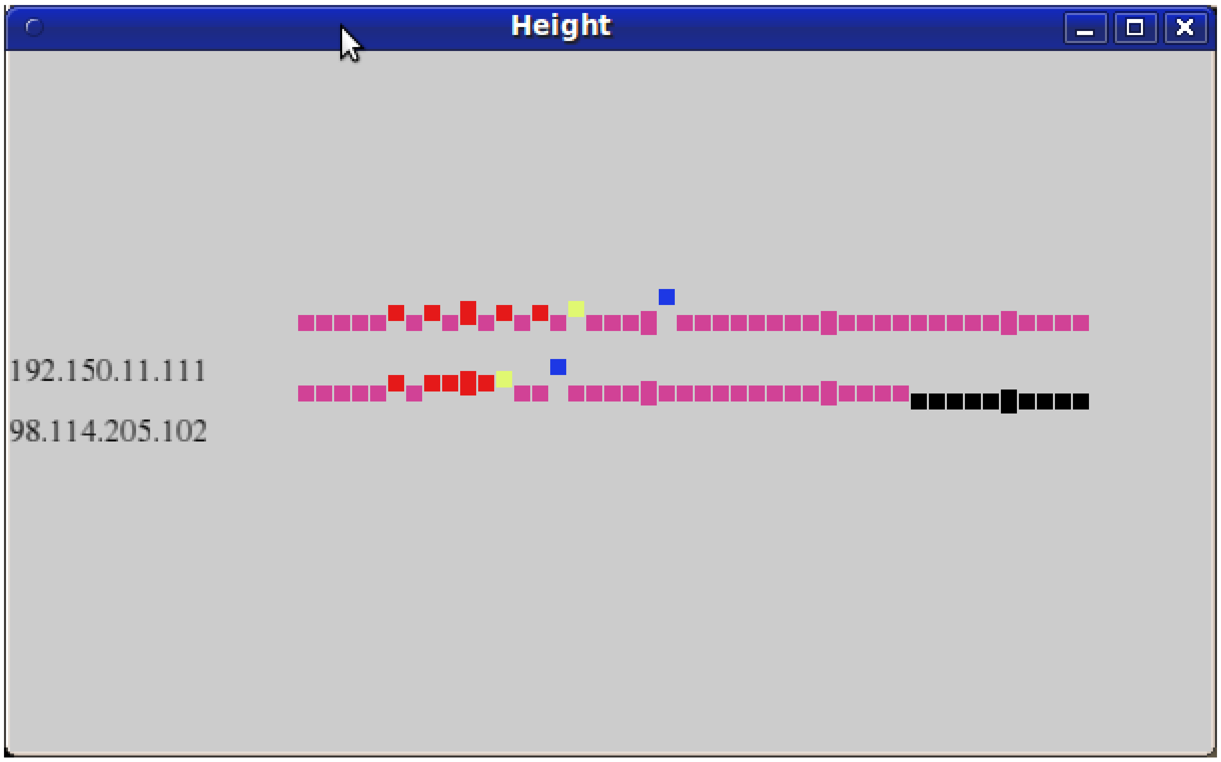


Figure 13: Figure of a height-differentiated graph

In this window the different user-defined groupings are indicated by both color and height. As in the color-differentiated graph, the IP addresses are on the left, and every tenth point is larger.

Creating a Parser

Creating a parser is as simple as creating a new Python script. To describe this process, it is best to start by looking at an event object, mainly its initialize function.

```
def __init__(self,t,h,e):
        self.Timestamp = t
        self.Host = h
        self.Event = e
        self.Color = [0.0,0.0,0.0]
        self.Height = 0.0
        self.EventHeight = 0.0
```

For the actual parser, only the first three data members are important: the Timestamp, the Host, and the Event. The Timestamp is for ordering, and can be anything, ranging from the actual timestamp, to packet number. The Host is the source of a piece of data, assumed for most data types to be a host IP address, but it can be anything. Finally, the Event is the string to match keywords with, it is what the matching implementation searches through to see if an Event matches a group. The color, height, and event height are discussed further in this paper.

The initialize function takes only three parameters, the timestamp, the source for a piece of data, and the event string. To create a parser, one simply needs to create a function that separates their data into those three components, creates a list of Events, and returns this list. For example, if we have data source that outputs the following:

```
'Line Num', 'Source IP', 'Keystrokes'
1, 192.168.13.4, ls
2, 192.168.13.4, cd test_dir
…
```

One would write a Python script that would run the following pseudo code:

example_parser.py

```
def example_parse():
    data_source = open('data_file')
    event_list = []
    for line in data_source:
        data_array = line.split(',')
        temp_event = Event(data_array[0], data_array[1], data_array[2])
        event_list.append(temp_event)
    return event_list
```

In this example the parser reads in every line from a file, splits each line into an array, using commas as a delimiter. It then creates a temporary Event, using this array, and appends that temporary Event to a list of Events.

Changing Parsers

Changing which parser to use with the visualizer is a trivial task. Using the example parser in the previous section, all one simply needs to do is:

1. Import the parser script into the visualizer
2. Change line 77 of the Visualizer script to call your parser script's parsing function

```
Change:
#import sebek_parser
to:
#import example_parser
and change line 77:
source_data = sebek_parser.get_data()
to:
source_data = example_parser.example_parse()
```

Grouping Files

To properly match a group of keywords with an Event, one must first create a Grouping file. To begin with, it is instructive to examine a Group object:

```
class Group:
    def __init__(self,c,k,h):
        self.Color = c
        self.Keywords = k
        self.Height = h
```

Each Group object has the following data members: Color, which is what color to display if matched, Keywords is a list of keywords to search an Event object's string for, and the Height is what height to displace a point if an Event matches a keyword.

Rather than have users hardcode a list of Group objects, the visualizer reads in a simple file called 'grouping' by default which consists of the following:

```
red channel, green channel, blue channel, height
keyword 1, keyword 2, keyword 3,..., keyword n
```

Colors are represented by their RGB value, going from 0 to 1, and the height should be constricted from 0.0 to 0.2. Each entry for both color/height and keywords are separated by a comma. Each keyword is compiled as a regular expression, so no spaces should follow each comma. If no color/height is specified, that is, four doubles, separated by commas, a random value is assigned for each color channel and height, for that line. An example grouping file might look like the following:

```
0.0,0.0,1.0,0.02
ls,ls.

ps,top
```

In the above example the regular expressions 'ls' and 'ls.' are considered one group, and if they match an event, the point in the visualizer will be displayed as a blue color, and will be displaced .02 upward from the default height for its source. The group consisting of 'ps' and 'top' will be assigned essentially a random color, and a random displacement.

Appendix B: Source Code

This code is also available by emailing kpgalloway@gmail.com

Event.py

```
class Event:
    def __init__(self,t,h,e):
        self.Timestamp = t
        self.Host = h
        self.Event = e
        self.Color = [0.0,0.0,0.0]
        self.Height = 0.0
        self.EventHeight = 0.0
    def show(self):
        print str(self.Timestamp) + ": " + str(self.Host)
    def uid(self,u):
        self.UID = u
    def pid(self,p):
        self.PID = p
    def fd(self,f):
        self.FD = f
    def ino(self,i):
        self.INO = i
    def com(self, c):
        self.COM = c
    def get_event(self):
        return self.Event
    def get_host(self):
        return self.Host
    def set_color(self,c):
        self.Color = c
    def get_color(self):
        return self.Color
    def set_height(self,h):
        self.Height = h
    def get_height(self):
        return self.Height
    def set_EventHeight(self,eh):
        self.EventHeight=eh
    def get_EventHeight(self):
        return self.EventHeight
    def get_r(self):
        return self.Color[0]
    def get_g(self):
        return self.Color[1]
    def get_b(self):
        return self.Color[2]
```

Group.py

```
class Group:
    def __init__(self,c,k,h):
        self.Color = c
        self.Keywords = k
        self.Height = h
    def show(self):
        print self.Color
        print self.Event
    def get_color(self):
        return self.Color
    def get_keywords(self):
        return self.Keywords
    def get_height(self):
        return self.Height
```

Size.py

```
class Sizes:
    def __init__(self):
        self.Point_Size = 8.0
        self.Space_Size = 1.0
    def Change_Point(self,change):
        if (self.Point_Size + change) > 0:
            self.Point_Size+=change
    def Change_Size(self,change):
        if (self.Space_Size + change) > 0:
            self.Space_Size+= change
    def get_Point(self):
        return self.Point_Size
    def get_Space(self):
        return self.Space_Size
```

Draw.py

```
#!/usr/bin/python2.6
from pyglet.gl import *


def draw_height(cam_test,matched,mins,maxs,window,p_size,s_size):
    lengths = { }
    for i in matched:
        if mins[i] < 0:
            mins[i] = 0
            maxs[i] = 44
        if maxs[i] < 0:
            maxs[i] = 44
        if mins[i] == maxs[i]:
            mins[i] = maxs[i] - 1
    glLoadIdentity()
    gluLookAt(cam_test.get_xyz()[0],cam_test.get_xyz()[1],1.0, #camera location
        cam_test.get_xyz()[0],cam_test.get_xyz()[1],0.0,       #what it's looking at
        0.0,1.0,0.0)          #point to camera
    glMatrixMode(GL_MODELVIEW)
    glColor3d(0.4,0.1,0.8)
    glPointSize(p_size)
    glLoadIdentity()
    total = 0.0
    #draw list of points
```

```
        glBegin(GL_POINTS)
        for i in matched:
          temp_pos = -1.9
          for j in range(mins[i],maxs[i]):#matched[i]:
                #print str(i) + ': ' + str(j)
                cam_x = cam_test.get_xyz()[0]
                if (j+1)%10 == 0:
                        glColor3d(matched[i][j].get_r(),matched[i][j].get_g(),matched[i][j].get_b())
                        glVertex3d(temp_pos/(float(window.width)/float(window.height)),matched[i][j].get_height()
+0.01+matched[i][j].get_EventHeight(),1.0)
                        glVertex3d(temp_pos/(float(window.width)/float(window.height)),matched[i][j].get_height()-
0.01+matched[i][j].get_EventHeight(),1.0)
                if j >= len(matched[i]):
                        j = len(matched[i])-1
                glColor3d(matched[i][j].get_r(),matched[i][j].get_g(),matched[i][j].get_b())
                x_coord = temp_pos/(float(window.width)/float(window.height))
                glVertex3d(x_coord,matched[i][j].get_height()+matched[i][j].get_EventHeight(),1.0)
                temp_pos+=0.030 * (float(window.width)/float(window.height))* s_size
        glEnd()
        glMatrixMode(GL_PROJECTION)


def draw_bars(cam_test,matched,mins,maxs,window,p_size,s_size):
        lengths = {}
        for i in matched:
          if mins[i] >= len(matched[i]):
                mins[i] = len(matched[i])-1
                maxs[i] = mins[i]+1
          if maxs[i] >= len(matched[i]):
                maxs[i] = len(matched[i])-1
                mins[i] = maxs[i]-44
          if mins[i] < 0:
                mins[i] = 0
        glLoadIdentity()
        gluLookAt(cam_test.get_xyz()[0],cam_test.get_xyz()[1],1.0, #camera location
          cam_test.get_xyz()[0],cam_test.get_xyz()[1],0.0,      #what it's looking at
          0.0,1.0,0.0)          #point to camera
        glMatrixMode(GL_MODELVIEW)
        glColor3d(0.4,0.1,0.8)
```

```
	glPointSize(p_size)
	glLoadIdentity()
	total = {}
	#draw set of points
	glBegin(GL_POINTS)
	for i in matched:
	  temp_pos = -1.9
	  for j in range(mins[i],maxs[i]):
		cam_x = cam_test.get_xyz()[0]
		if (j+1)%10 == 0:
			glColor3d(matched[i][j].get_r(),matched[i][j].get_g(),matched[i][j].get_b())
			glVertex3d(temp_pos/(float(window.width)/float(window.height)),matched[i][j].get_height()
+0.01,1.0)
			glVertex3d(temp_pos/(float(window.width)/float(window.height)),matched[i][j].get_height()-
0.01,1.0)
		glColor3d(matched[i][j].get_r(),matched[i][j].get_g(),matched[i][j].get_b())
		glVertex3d(temp_pos/(float(window.width)/float(window.height)),matched[i][j].get_height(),1.0)
		temp_pos+=0.030 * (float(window.width)/float(window.height)) * s_size
	glEnd()
	glMatrixMode(GL_PROJECTION)
```

Visualizer.py

```
#!/usr/bin/python2.6
import array
import random
import event
from group import Group
from size import Sizes
import get_timestamp
import parse_pcap_csv
import kdd_parse
import draw
import re
from pyglet.gl import *
from pyglet.window import key
from pyglet import font
from pyglet import image
import ctypes
```

```
'''Format for group file:
r,g,b,a:each goes from 0.0-1.0, if blank will randomly assign color
c,d,e,f : each comma can be a user-specified separator
 '''

max_dot = 10
min_dot = 0

point_size = 8.0
space_size = 1.0
size = Sizes()
#This is to get the groups
group_file = open('grouping.bak')
groups = group_file.readlines()
group_list = []
matched = {}
ip_heights = {}
#separate each group of colors and keywords
for i in range(0,len(groups),2):
        color_str = groups[i].strip('\n').split(',') #get the color of each one
        if len(color_str) < 4 or len(color_str) > 4:
          color = [random.random(),random.random(),random.random()]
          temp_height = random.randrange(99) * 0.001
        else:
          color = [float(color_str[0]),float(color_str[1]),float(color_str[2])]
          temp_height = float(color_str[3])
        events = groups[i+1].strip('\n').split(',') #separate each event to write to file
        temp = Group(color,events,temp_height)
        group_list.append(temp)
#This is for matching groups with events
source_data = get_timestamp.get_year()
current_height = 0.0
#Match events and compile REs
total=0
group_color = {}
group_height = {}
test_re = []
```

```
for i in group_list:
        for j in i.get_keywords():
          temp = re.compile(j)
          test_re.append(temp)
          group_color[j] = i.get_color()
          group_height[j] = i.get_height()

for i in source_data:
        if i.get_host() not in ip_heights:
          ip_heights[i.get_host()] = current_height
          current_height+=0.2
        if i.get_host() not in matched:
          matched[i.get_host()] = []

#set heights and colors
#match keywords with event data
for i in source_data:
        i.set_height(ip_heights[i.get_host()])
        for j in test_re:
          temp_result = j.search(i.get_event())
          if temp_result is not None:
                i.set_color(group_color[j.pattern])
                i.set_EventHeight(group_height[j.pattern])
        matched[i.get_host()].append(i)

ip_labels = []
gl_x = -0.5
#create the source labels
for i in ip_heights:
        temp_x = ((gl_x+1)/2)*600
        temp_label = pyglet.text.Label(i, font_name='Times New Roman',color=(15,15,15,255),
                                                                font_size=12,x=-0.5,
                                                                y=((ip_heights[i]+1)/2)*300,
                                                                anchor_x='left', anchor_y='bottom')
        ip_labels.append(temp_label)

group_labels = []
```

```
gl_x = -0.9
gl_y = 0.9
for i in group_list:
        temp_x = ((gl_x+1.02)/2)*600
        temp_y = ((gl_y+0.96)/2)*350
        temp_label = pyglet.text.Label(str(i.get_keywords()),font_name='Times New Roman',
          color=(15,15,15,255),
                                                                    font_size = 12,x=temp_x,y=temp_y,
                                                                    anchor_x='left',anchor_y='bottom')
        group_labels.append(temp_label)
        gl_y-=0.1

class Camera:
        def __init__(self):
          self.xyz = [-0.6,0.0,1.0]
          self.max_dot = 45
          self.min_dot = 0
        def move_x(self,offset,dot):
          #self.xyz[0]+=offset
          self.min_dot+=dot
          self.max_dot+=dot
        def move_y(self,offset):
          self.xyz[1]+=offset
        def move_z(self,offset):
          self.xyz[2]+=offset
        def get_xyz(self):
          return self.xyz
        def set_max(self,max):
          self.max_dot = max
        def get_max_dot(self):
          return self.max_dot
        def set_min(self,min):
          self.min_dot = min
        def get_min_dot(self):
          return self.min_dot

global mins
```

```
mins = {}
global maxs
maxs = {}
for i in matched:
        print 'Global: ' + str(i)
        mins[i] = 0
        maxs[i] = 44
#pyglet input
window = pyglet.window.Window(600,350,resizable=True)
window.set_caption('Color')
window2 = pyglet.window.Window(600,350,resizable=True)
window2.set_caption('Height')
window3 = pyglet.window.Window(600,350,resizable=True)
window3.set_caption('Key')
label = pyglet.text.Label('TESTING', font_name='Times New Roman',
                    font_size=56, x=2,
                    y=2
                    )
keymap = key.KeyStateHandler()
window.push_handlers(keymap)
window2.push_handlers(keymap)
window3.push_handlers(keymap)
label_test = pyglet.text.Label()
cam_test = Camera()
#OpenGL tests
#OpenGL initialize
glClearColor(0.8,0.8,0.8,0.0)

#draw the color differentiated window
def on_draw():
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        #camera test
        glMatrixMode(GL_PROJECTION)
        glPushMatrix()
        draw.draw_bars(cam_test,matched,mins,maxs,window,size.get_Point(),size.get_Space())
        glPopMatrix()
        glMatrixMode(GL_PROJECTION)
        for i in ip_labels:
          i.draw()
        #glPopMatrix()

#draw the height differentiated window
def on_draw2():
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glMatrixMode(GL_PROJECTION)
        glPushMatrix()
        draw.draw_height(cam_test,matched,mins,maxs,window,size.get_Point(),size.get_Space())
        glPopMatrix()
        for i in ip_labels:
          i.draw()

#draw the key
def text_window():
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
        glMatrixMode(GL_PROJECTION)
        glPushMatrix()
        glLoadIdentity()
        temp = 0.9
        glPointSize(8.0)
        for i in group_list:
          if temp < window.height and temp > 0:
                glBegin(GL_POINTS)
                glColor3d(i.get_color()[0],i.get_color()[1],i.get_color()[2])
                glVertex3d(-.9,temp,0.0)
                glEnd()
                temp-=0.1
        glPopMatrix()
        for i in group_labels:
          i.draw()

def on_key_press(symbol,modifiers):
        if symbol == key.Q:
          exit()
```

```
def update(dt):
        if keymap[pyglet.window.key.RIGHT]:
          for i in mins:
                  mins[i]+=1
                  maxs[i]+=1
          cam_test.move_x(0.00,1)
        if keymap[pyglet.window.key.LEFT]:
          for i in mins:
                  mins[i]-=1
                  maxs[i]-=1
          cam_test.move_x(-0.00,-1)
        if keymap[pyglet.window.key.UP]:
          cam_test.move_y(0.03)
        if keymap[pyglet.window.key.DOWN]:
          cam_test.move_y(-0.03)
        if keymap[pyglet.window.key.W]:
          size.Change_Point(1.0)
        if keymap[pyglet.window.key.S]:
          size.Change_Point(-1.0)
        if keymap[pyglet.window.key.A]:
          size.Change_Size(-0.1)
        if keymap[pyglet.window.key.D]:
          size.Change_Size(0.1)

window.on_draw = on_draw
window.on_key_press = on_key_press
window.switch_to()
glClearColor(0.8,0.8,0.8,0.0)
window2.on_draw = on_draw2
window2.on_key_press = on_key_press
window2.switch_to()
window.set_location(0,0)
window2.set_location(0,500)

glClearColor(0.8,0.8,0.8,0.0)

window3.on_draw = text_window
```

```
window3.on_key_press = on_key_press
window3.set_location(600,0)
window3.switch_to()
pyglet.clock.schedule_interval(update,1.0/60.0)
pyglet.app.run()
```

References

1. Bade, R., Schlechtweg, S., and Miksch, S. 2004. Connecting time-oriented data and information to a coherent interactive visualization. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Vienna, Austria, April 24 - 29, 2004). CHI '04. ACM, New York, NY, 105-112. DOI= http://doi.acm.org/10.1145/985692.985706
2. Kayacik, H. G. and Zincir-Heywood, A. N. 2006. Using self-organizing maps to build an attack map for forensic analysis. In Proceedings of the 2006 international Conference on Privacy, Security and Trust: Bridge the Gap between PST Technologies and Business Services (Markham, Ontario, Canada, October 30 - November 01, 2006). PST '06, vol. 380. ACM, New York, NY, 1-8. DOI= http://doi.acm.org/10.1145/1501434.1501474
3. Shabtai, A., Klimov, D., Shahar, Y., and Elovici, Y. 2006. An intelligent, interactive tool for exploration and visualization of time-oriented security data. In Proceedings of the 3rd international Workshop on Visualization For Computer Security (Alexandria, Virginia, USA, November 03 - 03, 2006). VizSEC '06. ACM, New York, NY, 15-22. DOI= http://doi.acm.org/10.1145/1179576.1179580
4. Viégas, F. B., Golder, S., and Donath, J. 2006. Visualizing email content: portraying relationships from conversational histories. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (Montréal, Québec, Canada, April 22 - 27, 2006). R. Grinter, T. Rodden, P. Aoki, E. Cutrell, R. Jeffries, and G. Olson, Eds. CHI '06. ACM, New York, NY, 979-988. DOI= http://doi.acm.org/10.1145/1124772.1124919
5. Koike, H. and Ohno, K. 2004. SnortView: visualization system of snort logs. In Proceedings of the 2004 ACM Workshop on Visualization and Data Mining For Computer Security (Washington DC, USA, October 29 - 29, 2004). VizSEC/DMSEC '04. ACM, New York, NY, 143-147. DOI= http://doi.acm.org/10.1145/1029208.1029232
6. KDD-Cup 1999 Dataset{online}
   http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html
7. Marty, R, *Applied Security Visualization*, Addison-Wesley, 2008
8. Spitzner, L, *Honeypots—Tracking Hackers*, Addison-Wesley, 2002
9. Pyglet Python Library{online} http://www.pyglet.org