APPLYING INFORMATION VISUALIZATION TO
COMPUTER SECURITY APPLICATIONS

by

Robert Bruce Whitaker

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

_________________________________
Robert F. Erbacher
Major Professor

_________________________________
Xiaojun Qi
Committee Member

_________________________________
Chad Mano
Committee Member

_________________________________
Byron R. Burnham
Dean of Graduate Studies

UTAH STATE UNIVERSITY
Logan, Utah

2010
ABSTRACT

Applying Information Visualization to Computer Security Applications

by

Robert Bruce Whitaker, Master of Science
Utah State University, 2010

Major Professor: Robert F. Erbacher
Department: Computer Science

This thesis presents two phases of research in applying visualization to network security challenges. The first phase included discovering the most useful and powerful features in existing computer security visualizations and incorporating them into the AdviseAid visualization platform, an existing software package. The incorporation of such a complete feature set required novel resolution of software engineering, human factors, and computer graphics issues. We also designed additional novel features, such as plugin interfaces, allowing for rapid prototyping and experimentation with novel visualization features and capabilities. The second phase of the research focused on the development of novel visualization techniques themselves. These novel visualizations were designed and created within AdviseAid to demonstrate that the features of AdviseAid are functional and helpful in the development process, as well as to be effective in the analysis of computer networks in their own right.
ACKNOWLEDGMENTS

I wish to thank the many people who have helped me complete this work. First and foremost, I want to thank my major professor, Dr. Erbacher, who has patiently and willingly helped me through this process. If it were not for his encouragement from the start, I would not have even entered the graduate program. If it were not for his guidance and constant assistance, I never would have finished it. I also want to thank Dr. Mano and Dr. Qi for the support they have given me as a part of my committee, and to the faculty and staff of the Computer Science Department.

I wish to also thank my wife. She has been there with constant encouragement and gentle support and has helped me make the time to complete this. Without her by my side, I doubt I ever would have found the drive and motivation to make it through the long process of writing a thesis.

My mother and father have always been there for me from the day I was born, and without their help, I would have given up on this thesis long ago. I cannot count the number of calls that I have made to them to get specific help on various pieces of this thesis, and with all of my schooling in general.

I am also extremely grateful to my friends and fellow students that have worked on AdviseAid with me over the years. I want to specifically thank Steena Monteiro, who has made work and this whole process fun and who has given great advice over the years, and Stephen Miller, who has greatly helped me broaden my understanding of computer science, and has helped me see how exciting software development can be.

RB Whitaker
## CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>PREVIOUS WORK</td>
<td>8</td>
</tr>
<tr>
<td>The Visualization Design Process</td>
<td>8</td>
</tr>
<tr>
<td>Common Features of Security Visualizations</td>
<td>11</td>
</tr>
<tr>
<td>Visualization Development Platforms</td>
<td>18</td>
</tr>
<tr>
<td>Keys to Visualization Platform Design</td>
<td>22</td>
</tr>
<tr>
<td>METHODS AND IMPLEMENTATION</td>
<td>25</td>
</tr>
<tr>
<td>AdviseAid Overview</td>
<td>25</td>
</tr>
<tr>
<td>Input Processing</td>
<td>32</td>
</tr>
<tr>
<td>Data Processing</td>
<td>42</td>
</tr>
<tr>
<td>Visualization Framework</td>
<td>45</td>
</tr>
<tr>
<td>Visualizations</td>
<td>51</td>
</tr>
<tr>
<td>Other Components of AdviseAid</td>
<td>62</td>
</tr>
<tr>
<td>RESULTS AND CONTRIBUTIONS</td>
<td>75</td>
</tr>
<tr>
<td>Comparison of Features</td>
<td>75</td>
</tr>
<tr>
<td>Novel Visualization Techniques</td>
<td>76</td>
</tr>
<tr>
<td>Separation of Technique and Framework</td>
<td>76</td>
</tr>
<tr>
<td>Prototyping</td>
<td>77</td>
</tr>
<tr>
<td>Production-Level Development</td>
<td>78</td>
</tr>
<tr>
<td>FUTURE WORK</td>
<td>79</td>
</tr>
<tr>
<td>Rework of the Data Pipeline</td>
<td>79</td>
</tr>
<tr>
<td>Plugin System Improvements</td>
<td>81</td>
</tr>
<tr>
<td>Improved Mapping System</td>
<td>82</td>
</tr>
</tbody>
</table>
CONCLUSIONS..................................................................................................................85

CASE STUDY: THE TRIANGLE VISUALIZATION .........................................................88

  Introduction..................................................................................................................88
  Previous Work ...........................................................................................................91
  Methods and Implementation .....................................................................................94
  Results .......................................................................................................................106
  Future Work and Conclusions ..................................................................................113

REFERENCES .............................................................................................................117
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>AdviseAid architecture diagram.</td>
<td>27</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Database Mapper Utility screenshot</td>
<td>41</td>
</tr>
<tr>
<td>Figure 3</td>
<td>The mappings GUI</td>
<td>45</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Geographical mapping in VisAlert</td>
<td>48</td>
</tr>
<tr>
<td>Figure 5</td>
<td>An example topological map</td>
<td>49</td>
</tr>
<tr>
<td>Figure 6</td>
<td>A screenshot of the Toplogy Forge</td>
<td>50</td>
</tr>
<tr>
<td>Figure 7</td>
<td>VisAlert screenshot</td>
<td>53</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Triangle visualization screenshot</td>
<td>55</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Pie Chart (advanced) screenshot</td>
<td>56</td>
</tr>
<tr>
<td>Figure 10</td>
<td>ForceVis screenshot</td>
<td>57</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Simple Pie Chart visualization screenshot</td>
<td>59</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Filtered data on a Simple Pie Chart visualization</td>
<td>60</td>
</tr>
<tr>
<td>Figure 13</td>
<td>The drilldown display</td>
<td>61</td>
</tr>
<tr>
<td>Figure 14</td>
<td>The Plugin Manager interface</td>
<td>65</td>
</tr>
<tr>
<td>Figure 15</td>
<td>The image and video capturing interface</td>
<td>70</td>
</tr>
<tr>
<td>Figure 16</td>
<td>Video and image capture manager screenshot</td>
<td>71</td>
</tr>
<tr>
<td>Figure 17</td>
<td>A screenshot of the current mapping system GUI</td>
<td>83</td>
</tr>
<tr>
<td>Figure 18</td>
<td>A basic ternary plot</td>
<td>89</td>
</tr>
<tr>
<td>Figure 19</td>
<td>A screenshot of the Triangle visualization</td>
<td>91</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Illustration of the overall architecture of the Triangle visualization</td>
<td>95</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Screenshot of zones in the triangle visualization</td>
<td>104</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Two port scans in the Triangle visualization</td>
<td>107</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>23</td>
<td>Anomalous activity shown in the Triangle visualization</td>
<td>108</td>
</tr>
<tr>
<td>24</td>
<td>The Triangle visualization mapped to packet size</td>
<td>109</td>
</tr>
<tr>
<td>25</td>
<td>The Triangle visualization with video and web traffic</td>
<td>110</td>
</tr>
</tbody>
</table>
INTRODUCTION

Computers are used throughout the world, in nearly endless ways. The United States Census Bureau indicates that in 2007, 67.1% of people age three and above in the United States live in a home with Internet access, and the number continues to grow [34]. But as computer use has increased, so has computer misuse. Computer attacks have become more and more common [29]. The CSI Computer Crime and Security Survey for 2008 reported that in 2008, 43% of businesses that responded to their survey had experienced a computer security incident, and 26% had experienced more than ten [25]. Further, the monetary cost of these attacks is high. Of the attacks reported in 2008, the average loss caused by the attack was $289,000 [25].

In order to stay safe from these attacks, individuals, companies, and other organizations have needed to take action to protect themselves. As the variety and frequency of attacks has increased, so has the need for additional tools and systems designed to protect computers. Indeed, because of the dynamic and complex nature of computer-based attacks, it is likely this problem will never have a final, definitive solution.

There is a wide variety of types of tools and systems being used in computer security. Virus scanners, spam filters, and firewalls are all widely used [25]. More advanced systems, including intrusion detection systems such as Snort, are being put to use as well, through to a lesser degree [25]. Some companies and research groups are using proactive measures like honeypots to try to determine specifically what it is attackers are trying to do, while others still attempt to use data mining and genetic
algorithms [21]. It is clear that in order to provide safety to a network or computer, its protections must be as widely diverse as the field of potential attacks.

One type of security tool that has shown great potential is that of visualization. Visualization has become a popular and effective method for detecting and addressing security concerns. The value of and need for visualization in computer security is so great that the VisSEC workshop and community was formed around it. Visualization is a field of computer science wherein data is displayed on the screen graphically in a way that allows a system administrator or analyst to interpret the visual display of the data much faster than textual representation of the same data. The complexity of visualization can range greatly, from basic plots and charts such as bar and line graphs, to more powerful and complex methods that involve animation, data filtering, and a variety of changing visual attributes. Basic visualizations can display a limited amount of information but remain easy to understand, while complex visualizations can display more information, usually at the cost of being less intuitive.

In recent years, computers and networks have seen an explosion in the amount of data they process, transmit, and store. The sheer amount of data prevents a human from trawling through it to discover meaningful information, such as vital clues to security attacks and vulnerabilities. A wide variety of tools exist for reducing the amount of data an analyst must explore, such as Snort and other intrusion detection systems. These tools, however, are not enough. They often hide information of value by filtering it, while in many cases, leaving the user with far too much information to process. The human visual cortex can process images much faster than text, and applying information
visualization to this problem allows humans quickly gain insight into computer and network data. Ware [37] describes several reasons why visualization is beneficial. He indicates that visualization allows users to comprehend vast amounts of data, because of the parallel nature of the human visual cortex. Also, unlike information calculated algorithmically by automated tools, visualization allows users to perceive emergent properties of the data set not otherwise anticipated. It can reveal problems with the way the data was collected, facilitates an understanding of large and small scale features of the data, and helps users to form hypotheses.

Throughout this thesis, visualization refers to information visualization, rather than scientific visualization. Information visualization is the process of displaying data that has no physical form, whereas scientific visualization seeks to replicate a physical phenomenon. The goal of information visualization is to provide the best method for discovering abstract information, while scientific visualization values realism.

There are a wide range of visualization techniques used in computer security today, and many more in development. The goal of these visualizations is to take low level network data, or information collected on a single machine, and display it on the screen in a manner that will allow an analyst to quickly identify important security events, such as attacks or attempted attacks. Such visualization techniques bypass the time-consuming and tedious process of manually sifting through the low level data, and they are faster than tools that process the data for viewing its text-based or table-based raw output.
North [22] points out that there are no natural constraints on the types of visualizations that can be created, so the possibilities for visualization techniques are limitless. In regards to comparing visualization techniques, Ziemkiewicz et al. state that it is widely recognized that any given method will be better for some applications, and worse for others [40]. Consequently, out of the large number of possibilities, only a small number of techniques actually help analysts. Further, because visualizations are often only suited to a particular task, we can determine that there is a significant need to use multiple visualizations concurrently. Additionally, because so many possible visualizations can be created, but relatively few will be helpful, it becomes imperative to create an ability to rapidly test new visualization techniques.

Most visualization techniques are derived from abstract representations of data, such as graphs, common plots, and charts, including bar charts and scatter plots [15]. Components of the data set are mapped to visual attributes, such as color, shape, and size. These kinds of visualizations represent data in an abstract manner, wherein users can monitor the basic visual attributes of a visualization, with the understanding that when the visualization or part of a visualization takes on a certain appearance, a problem has occurred.

It is important to note that many visualization techniques share common features. For example, all visualizations must process a data set in one form or another. This requires each visualization tool to implement file parsers, create database connections, or read from a data set in another form. Many security-based visualizations read from the same type of data, such as TCP dump files, Snort logs, and CSV files. Also,
visualizations usually need handle data sequentially, requiring an iterative event loop in which data is handled in short time intervals. Filtering and drilldown are also commonly needed in visualization techniques. Screenshots and video capturing are frequently used to produce reports, as well.

Because visualizations have so much common ground, it would be of great benefit for researchers, developers, and analysts to have access to a platform that provides many of the common requirements of visualizations, makes it easy to write new visualization techniques, and provides mechanisms for extending the platform in the event that additional components are desired. A few such platforms exist [33, 36, 38], though they are limited either in terms of flexibility in the types of visualizations that can be created or by the lack of features needed to fully allow users to import, manage, and map to data sets.

AdviseAid is an existing visualization platform designed to make visualization prototyping simple and easy. It provides a data pipeline to the visualizations, giving them access to a wide variety of data sources, gives the user the ability to filter the data set, provides visualizations with the ability to drill down into the underlying data set, and to communicate with other visualizations. It allows developers to easily implement new visualization techniques. AdviseAid also provides a plugin system, allowing developers to extend the platform as needed. AdviseAid is powerful enough that the visualization’s developer can go beyond the prototyping phase and produce production-caliber visualizations.
These attributes allow AdviseAid to greatly help in the field of applying visualization to security problems. Because of its simplicity and power, AdviseAid can allow developers and researchers to quickly create, test, and deploy new visualization techniques. Because the AdviseAid platform takes care of many of the common aspects of development process, developers are able to concentrate solely on developing their visualization. This also gives developers more time to address problems or limitations with the individual visualization’s design and otherwise improve the visualization technique.

For the design and development of AdviseAid, Dr. Erbacher provided the high level direction for needs and requirements, while I was responsible for researching needed the needed capabilities, designing user interface components, designing the software architecture, and guiding the development process with feedback from and discussion with Dr. Erbacher.

The remainder of this thesis is structured in the following way. In the Previous Work section, we analyze and evaluate the common features that many security visualizations, including AdviseAid, need. This section also takes a brief look at other visualization platforms, such as InfoVis [36] and Improvise [38]. In the Methods and Implementation section, we look at the design of AdviseAid and discuss the features of AdviseAid that were created. In the Results and Contributions section, we look at how the AdviseAid platform helps to expand the field of visualization in computer security. Next, we discuss future steps of AdviseAid’s development in the Future Work section and what AdviseAid has shown us so far in the Conclusions sections. The final section
of this thesis is a case study of a novel visualization technique, the triangle visualization, which we designed and created to test and demonstrate the capabilities of AdviseAid, as well as to be a useful new visualization in its own right.
The goal of the AdviseAid framework is to provide a common set of tools that visualizations, specifically security-related visualizations, can utilize to streamline their development. We took into account three important aspects of computer security visualizations as we assessed the needs of the AdviseAid platform. First, we analyzed the visualization design process. Second, we looked at a cross-section of the many security visualization techniques in existence in order to determine which features were commonly implemented, or that their creators indicate would be useful additions. Third, we looked at the existing platforms for creating visualizations, and analyzed needs for improvement.

The Visualization Design Process

Designing a visualization is a complex process involving a wide variety of aspects. Visualization design draws on many fields, including computer graphics, mathematics and statistics, cognitive psychology, art, and GUI design, along with the domain experts that drive the creation of new visualization techniques [6]. To aid in the development of new techniques, a general process for creating visualizations has begun to be established. It is quite common for visualization designers to draw from nature, architecture, and art.

The field of information visualization is fairly new, and as a result, little formal information about the design and needs of a new visualization exists. Craft and Cairns [4] point out that while there are many publications regarding the design of specific techniques, there is little about the overall design process, and that the people seeking to
create a new visualization technique have little to draw on. The authors also discuss how the visualization community is currently seeking to resolve this issue.

A handful of works give a brief overview of the general process of visualization design and development [22, 28, 37]. Each of these methods describes a variation on the following four basic steps. First, input data is read in and stored in a standardized form. Second, the data is transformed to glyphs, wherein various components of the data may modify the appearance of the glyphs in a variety of methods. Third, the glyphs are displayed on the screen using the computer’s windowing and rendering system. Finally, the human visual cortex reverses the transform to provide the user with an understanding of the data in the data set. The user is then able to interact with the visualization to modify it in various ways, and indeed, this interaction is so important that it is almost a necessity for visualizations. The completed design of the visualization must take all of these components into account.

North [22] provides an overview of the general process. He indicates that there are two major challenges for any visualization that must be overcome: complexity, meaning the visualization must work for a variety of forms of datasets; and scalability, meaning the visualization must handle significantly large data sets, both in terms of algorithmic complexity, as well as in terms of the ability to display information about a large data set in a way that is comprehensible by humans.

In regards to displaying large datasets, Shneiderman [28] proposes that an overview of the entire data set, along with the user’s ability to drill down to smaller aspects of the data set and filter out unnecessary data, is among the most effective
methods for providing the user with scalability. This leads to the commonly quoted mantra “overview first, zoom and filter, then details on demand.” North also discusses the principle of using overviews to deal with large datasets [22]. He points out that an overview gives the viewer the ability to see the big picture, while details-on-demand is then required to view small scale aspects of the data.

Several interaction strategies are discussed throughout the literature. Interaction with the visualization allows the user to discover more information about the data in a variety of ways. North lists some of the most useful and common interaction techniques [22]. This list includes selection, linking to interactively relate information between multiple views, filtering, rearranging, and remapping.

The process of transforming assembled data into glyphs is one of the most important parts of the process. It is this step that gets the data to a form that the human eye will comprehend, once it is displayed. Glyphs are any form of marker that represents a piece of data. These glyphs come in a variety of forms, such as points or other simple shapes, lines and curves, regions such as polygons and volumes, or icons. Relationships between data items can be represented as other additional glyphs. Attributes of the data are mapped to various visual properties of the entity’s glyph. These properties commonly include position, color, shape, orientation and size, but can also include additional fields like texture, transparency, motion, density, and so on. The key principle of the mapping process is to match the important data attributes to the most visible visual properties [22]. Position is the most significant visual property in most cases, while visual properties depend on the type of data being analyzed. There are some systems that attempt to
automatically determine mappings [26], though this can limit the capabilities of the system, based on the restrictions needed to allow for automated mapping detection. Most visualizations ultimately need to allow the user to choose the necessary mappings, though many first passes at a new visualization technique have a specific set of mappings that is hard-coded, and cannot be changed.

While visualization techniques may be able to neglect some of the tasks described in these sections, most stand to benefit a great deal by utilizing them. Having discussed the known methods for designing visualizations, we next turn our attention to the way current visualization techniques function, especially those related to computer security, and discuss some of the common features that they share.

**Common Features of Security Visualizations**

To further understand what is required of visualizations in general, and computer security visualizations specifically, we surveyed the literature and analyzed common features that had been implemented, as well as features not implemented but that the designers still wanted to add in the future. This survey illustrates what is needed in a visualization, and assists in the decision making process for what is required of a visualization development platform. Because computer security visualization is a growing field, there are many visualization techniques in existence. Therefore, the goal of the survey was to get a broad cross-section of the existing types of security visualizations, rather than catalog all existing techniques.

Our survey included twenty visualizations, described briefly below:
1. **VISUAL.** Displays communication patterns between a local network and external hosts [2].

2. **Spinning Cube of Potential Doom.** Plots network traffic within a cube, with source IP, destination IP, and port as the three axes [15, 16].

3. **SecureScope.** A 3D tool for situational awareness, correlating bar graphs of attacks with geographical location, business components, and time [27].

4. **PortVis.** Analyzes IP protocol data from a table, and displays information within several plots, and is used to detect port scans, and suspicious traffic patterns [18].

5. **NvisionIP.** Processes NetFlow data, and provides situational awareness using three levels of detail, ranging from entire class-B networks to individual machines [39].

6. **VisFlowConnect-IP.** Also uses NetFlow data to allow a user to visually assess the connectivity of large, complex networks [39].

7. **Network Eye.** Provides an end-to-end view of application communication across the internet, combining two visualization techniques [6, 7].

8. **Tudumi.** Provides a system for visualizing log data in various ways, with the ability to provide a summary of data and drilldown [31].

9. **SHriMP.** Helps people browse complex information spaces by providing multiple perspectives and nested interchangeable views [30].
10. **SnortView.** Displays Snort logs visually, combining two modules to handle log analysis, and a visual display. Also provides real-time monitoring capability [12].

11. **Mukosaka and Koike.** Combines logical, temporal, and geographical information into a single 3D visualization to display IDS logs, designed to monitor the user’s own internal network [20].

12. **IP Matrix.** A method of visualizing the entire IP address space, using an Internet view and a local view to process Snort logs, using a combination of scatter plots, heat maps, and histograms [13].

13. **StarMine.** Displays log files on a world map, with links to logical and chronological displays [9].

14. **Teoh et al.** An animated visualization designed to make it easier to detect and analyze anomalies using multiple representations to view Border Gateway Protocol (BGP) data [32].

15. **Visual Firewall.** Uses four different simultaneous views to display firewall and IDS alarm data, to allow users to see where their firewall may be poorly configured [17].

16. **MieLog.** An interactive visual log browser to process a generalized log file format in real time [31].

17. **Portall.** Gives system administrators a view of the processes using the network on a monitored machine, correlated with network activity that the processes are using [8].
18. **IDGraphs.** Processes NetFlow data to provide a flow-level trace plotted against time [24].

19. **SeeNet.** Displays a local network, along with the data associated with it, using a map of the network [3].

20. **SVision.** A 3D plot of hosts that move through a 3D space defined by the services that they use, allowing the user to visually cluster them into normal and abnormal categories [23].

We discovered many similarities among the way these techniques were implemented, along with the features that the creators indicated that they would like to add in the future. These similarities are discussed below.

**Architecture**

The architectures of most of the visualization techniques are quite similar. The visualizations read in data from a data source, typically a log file, and display it on the screen using some form of rendering system. The user interacts with the visualization in a variety of ways, and the architecture provides the user with the ability to filter, and perform drilldown.

Not all groups reported their choice of programming language, but of those who did, most indicated that Java was used [27, 30, 39], though C++ was mentioned as well [16, 27]. Many of the visualizations used OpenGL [16].

Nearly all of the visualizations use multiple visualization techniques simultaneously to allow the user to discover meaningful information. The vast majority of the techniques surveyed include multiple techniques (typically called “views”) to
display the data in various ways within the complete visualization [2, 12, 18, 20, 30, 39]. In most cases, this is done to allow for details-on-demand purposes, wherein the view of the full data set and the view of small details require different techniques. A number of other visualizations are a part of a larger collection of visualizations that work together [2, 7, 9, 27, 39]. Some of the sources surveyed state that their technique is not sufficient as a standalone technique to detect all possible attacks [31, 32], though we suspect that there are many other techniques that are this way as well. It is clear, though, that single visualization techniques are generally insufficient for all possible uses.

Inter-visualization communication, or at least, inter-view communication was another common thread we found. Of the visualizations that provide drilldown via a secondary visualization or view, most have a great deal of coupling between the visualizations [2, 12, 30, 39], for example, allowing one visualization or view to know when data is being selected in another.

Another common feature of the surveyed visualizations was the ability to animate the display by handling the data in a sequential manner and displaying it in distinct time steps. Some visualizations provide this mechanism already [3, 11, 32, 39] while others indicate this is a desirable feature to add in the future [2, 9, 12]. In a related aspect, some techniques, such as [16], want to have features to scale and reverse time, allowing users to have various playback speeds.

**Input**

Visualizations take data from data sources in one form or another. Visualizations that are designed to be general-purpose tend to read general-purpose data storage, like
CSV files, or XLS files. Additionally, visualizations geared toward computer security often wish read from more specific data sources, such as log files. This section discusses the types of input sources that our visualization survey covers.

Most of the visualizations surveyed read data from various logs, from raw network traffic, or processed network traffic. Takada et al. [31] and the Visual Firewall [17] both indicate that they read from log files, without specifying in detail which kind, while Snort logs are also quite common [12, 13, 20]. In addition to Snort logs, [17] handles preventia logs as well. The Spinning Cube of Potential Doom [15] reads log files from Bro IDS. StarMine [9] indicates that while currently they could only handle a single file type, they would like to be able to parse a variety including firewall logs, system logs, and packet sniffer logs. Tudumi [31] uses a general log format to access multiple types of data.

Ball et al., who produced VISUAL, want to add the ability to process network traffic at the packet level [2], while [24] and [39] utilize network traffic at the flow level, specifically the NetFlow file format.

Out of the visualizations we looked at, only one specifically mentioned a generic delimiter separated text file [18], though this type of feature would be more valuable to general-purpose visualizations. Additionally, only one visualization specifically mentioned connecting to a database [27], most likely because reading the original data in log files or traffic captures was a higher priority. This shows, however, that there is value in reading data from a database.
Aside from the type of file accessed, there are two other features of the input that are worth mentioning. Several reported that they would like to have the ability to read from multiple forms of data and to be able to read from them simultaneously [9, 31]. Because the visualizations surveyed are designed for computer security on a real network, many of these sources indicate that they would like to be able to display real-time data sets [2, 16, 39] to enable analysts to see the current state of their network, as opposed to “pre-recorded” data.

Data Manipulation

Once it is read from the data source, data is usually stored in memory and processed before being visualized. This is often an interactive process, wherein the user views the visualization and performs various tasks to change the data set ultimately displayed, and the visualization updates itself. We found a number of features to be common throughout most of the visualizations, including filtering, searching, and data mapping.

Filtering came up in nearly all of the visualizations. Filtering allows the user to restrict the data being viewed to focus on areas of interest. Some of the visualizations provide a significant filtering mechanism [39], others only have a limited mechanism for filtering [20], and still others only mention it as an important aspect that they would like to include in the future [16, 24].

For one visualization [24], searching was an important feature. This allows the user to locate where a specific set of data was being drawn within the visualization, and perform highlighting, as mentioned in [39].
A number of visualizations also discussed data mapping as an important aspect of their visualization [11, 17, 18]. Data mapping allows the user to pair parts of the data to various visual attributes, giving the visualization greater flexibility because it can be used to represent the data in a variety of different ways.

**Visualization Design**

Much of the design of a visualization is specific to the technique itself, but there are some similarities among the different visualizations. A few of these are discussed below.

Almost without exception, drilldown was utilized [2, 16, 18, 39]. While the interaction techniques for determining what specific items were chosen for drilldown, the need is there for virtually all visualizations.

Another aspect of the visualization design that was common among multiple visualizations was that of maps. For example, many security visualizations want to be able to locate an IP address either topologically [3] or geographically [9, 13, 20].

**Visualization Development Platforms**

Our survey clearly demonstrates that visualizations tend to follow similar design and development lifecycles, and that the security visualizations in existence currently share a great deal of common features. Because of this, it would greatly simplify the process of designing and implementing visualizations if they could each share a common codebase that provides these features to the visualization developer. Such action would enable the visualization developer to concentrate solely on a specific visualization
technique without needing to worry about the complexities of the complete process. Indeed, this is the function of AdviseAid.

AdviseAid, however, is not the only platform currently in existence that is available for developing visualizations. Before discussing AdviseAid, it is helpful to analyze other existing systems. In this section, we will look at the existing frameworks designed for creating visualizations.

One of the newer, more powerful visualization platforms in existence is IBM’s Many Eyes project [10]. Many Eyes is a web-based platform that includes a wide variety of visualization techniques. These techniques are mostly based on simple plots, including scatter plots, bar charts, bubble charts, line charts, pie charts, and geographical maps, such as a US county map or a world map. Input for Many Eyes is restricted to tab-delimited files with a maximum file size of five gigabytes. Many existing visualizations utilize their own special processing of the data, e.g., a US county map that recognizes state abbreviations like AZ, and TX. A key component of Many Eyes is that datasets are made available to the public, so users can discuss the data and share observations. Many Eyes however, is limited in its suitability for visualization development in several ways, especially for security-related data. While the system provides a vast array of visualization techniques, users are not able to produce their own visualization techniques. The system is also missing many important features that are useful to visualizations, such as drill down, or the ability to handle various file types. Many Eyes also produces only static graphs, which may miss key features of a security-related dataset. In addition,
because datasets are published to the world, most companies would likely be hesitant to place their network data in the public’s view.

Improvise [38] is another visualization development platform comparable to Many Eyes. While it is missing many of the features of Many Eyes, it provides a number of other tools that may be more helpful for visualizations in general. Improvise has a strong emphasis on coordinating multiple views, which is a feature that Many Eyes is lacking. The architecture and user interface allows users to build and browse through multiple customized visualizations without a high demand for programming skills. This makes it easy to produce novel visualization methods for data sets, but the platform does not easily lend itself to extending the types of visualizations available, even by a skilled developer. Improvise has a powerful filtering framework, allowing users to get access to the data they want without being cluttered by unnecessary data, along with a collection of other tools built into the GUI for working with the dataset. Improvise lacks any sort of feature that would allow for skilled developers with novel visualization ideas to be able to implement them and extend the framework.

The University of Indiana has also developed a platform for creating visualizations called the InfoVis Cyberinfrastructure (IVC) [36]. IVC is built on top of the Eclipse Rich Client Platform, giving it a large assortment of useful tools with little work. The platform allows developers to produce visualizations in different languages and using different drawing systems while keeping IVC a single “seamless” software package. IVC draws on the strong plugin system that is part of the Eclipse Rich Client Platform to make it easily extensible. The IVC is unique because it does not require data
to be in any particular format, neither does it need to be stored in any particular data structure. However, the IVC framework does not appear to provide defaults for these features to any significant degree, meaning that visualization developers must implement their own file parsers and data structures. This requires each visualization to be implemented independently from each other. While the end user will get the benefit of a single visualization package, the visualization developer will still need to spend time on a variety of components beyond the core visualization implementation.

Mondrian [33] is another visualization development platform whose strength is that multiple visualizations can be executing concurrently, and the different visualizations are “fully linked,” meaning that data selected in one visualization is marked in all other visualizations. Mondrian provides a fairly large collection of visualization techniques including common plots like bar charts and histograms, but also includes some less common graphs, such as SPLOM and Mosaic plots. Mondrian can only use standard tab-delimited files. Mondrian, like many of the other systems, does not explicitly allow visualization developers to add in their own new visualization techniques, though the package is open source.

One final platform worth noting is IVEE [1]. IVEE is designed to connect to databases, and tries to automate much of the visualization setup work for the user. IVEE provides a number of standard graph-based visualizations, and multiple visualizations can be active simultaneously. IVEE provides drilldown techniques, and the user can provide an HTML file to indicate to the system how the drilldown data should be displayed. IVEE also allows multiple clients running on different workstations to communicate with
each other. IVEE, too, though, only provides minimal support for new visualization developers, and instead concentrates on the end user.

**Keys to Visualization Platform Design**

While they provide a number of useful techniques and tools to the end user, the existing visualization platforms do not appear to provide many of the features needed to assist in the development of new visualization techniques. Typically, each visualization is developed as a separate component, including additional filtering and input tools as needed by the visualization, forcing developers to turn their attention from core visualization coding. Each of these platforms tend to provide a set of disassociated visualization techniques, along with glue code for the supporting framework, while many do not even provide adequate functionality for developers to even add new visualization techniques to the system. The platforms do not allow visualization designers to prototype a visualization quickly and with little investment.

An ideal visualization development platform, therefore, should include the following features:

1. *Ability to read from various data sources.* The visualization designer should not need to worry about reading from a data source. The platform should provide this to the visualization in a manner which is independent of the data source.

2. *Built in features for manipulating data before visualization.* Once data is read in from the data source, the user must have control over the form and values of data that reaches the visualization. This comes in the form of filtering and aggregating data, among other things.
3. **Support for large datasets.** Visualization is especially effective on large datasets, and as such, a visualization development platform must have an inherent ability to handle very large datasets without restrictions like those found in [10].

4. **Support for time and animation.** Many data sets occur over a long period of time, and as such, it is preferable for this data to be displayed in an animated form, rather than as a static graph of the accumulated results.

5. **A simple framework for creating a new visualization.** An ideal visualization development platform would allow the developer to focus solely on their visualization technique. The process of making the technique itself, however, should also be simplified as much as possible. This includes allowing development in a simple and powerful language, ideally in multiple languages, and using various rendering tools such as OpenGL, to allow the developer to use knowledge that they already have.

6. **A system for mapping data parameters to visual attributes.** Each visualization should not need to worry about what fields of a dataset should be mapped to the visual attributes. A single extensible GUI, provided by the framework, allows visualization developers to just use the current mapping, without needing to have each visualization implement their own mapping GUI.

7. **Utilization of multiple visualizations, and inter-visualization communication.** In order to be useful, many visualization techniques must work with other techniques. This is commonly done to provide an overview + detail visualization, wherein the overview and the detail visualizations can be created separately. It
also allows for multiple visualizations to work together to complement each other’s strengths. It is imperative that a visualization platform allow multiple visualizations to run simultaneously. Additionally, visualizations must be able to communicate with each other, at least to the extent that selected data in one visualization can be coordinated with selections in other visualizations.

8. **Data Drilldown.** In order for most visualizations to be useful, they usually require the user to be able to interact with the visualization to drill down to the underlying raw data. Search features would also be beneficial to the user.

9. **Extensibility.** In order for a visualization system to be most effective, the platform must be extensible. Visualizations, data source parsers, and other components need to be created without having to interact with or recompile the code for the platform, and added into the program dynamically in the form of a plugin system.

AdviseAid was designed with these basic principles in mind, and is intended to be a visualization platform that is useful to both the end users, and to the developers. These basic requirements allow developers to quickly and easily prototype a new visualization technique and test its usefulness. No system is currently designed to handle all such capabilities. One research goal, based on software engineering needs, was to design a novel environment and associated capabilities to integrate a larger set of these capabilities than currently exists in other platforms.
METHODS AND IMPLEMENTATION

The current version of AdviseAid has been in development since the fall of 2004. It is a fairly large program, with a huge variety of features and components. It is currently about 85,000 lines of Java code, and includes a wide variety of visualization techniques, most of which are novel, though a few are implementations of existing techniques, or basic plots are included.

This section describes the features that we implemented in AdviseAid. We start by providing a high level overview of the way AdviseAid works. Such an overview allows us to show the impact of our survey of previous work, and the research there engendered to develop a novel set of capabilities to more effectively meet the needs of security and network analysts. We then discuss the important or novel details of the AdviseAid framework, looking at the overall architecture, the input parsing, data handling, and visualization tools that we have created. We conclude this section by discussing the novel visualization techniques that have been produced.

AdviseAid Overview

AdviseAid is a software program and API designed to make visualization techniques easy to implement. One of the primary goals of AdviseAid is to allow a developer to prototype a new visualization technique, to determine quickly if a technique has value and is worth continued development time. AdviseAid is capable of producing production quality visualizations as well. Most visualizations to-date have been for the purposes of computer security, though more general purpose visualizations have been
developed more recently, and AdviseAid is quickly adapting to fill the needs of those visualizations.

AdviseAid is written in the Java programming language. This decision was made to allow for faster prototype development, as Java applications typically require less development time than those written with the other common languages, at the expense of a small decrease in performance speed. A variety of Java-related technologies are used in AdviseAid, including Swing, Java2D, and JOGL.

AdviseAid includes a large number of components that many visualizations need, including being able to read from a variety of input sources, including CSV files, databases, and live network traffic.

We now turn our attention to the basic architecture of AdviseAid. At its core, AdviseAid simply takes inputs from various sources, typically files, processes them, and hands the results to individual visualizations who determine individually how they will display the data. Figure 1 shows the basic architecture of AdviseAid.
Figure 1. AdviseAid architecture diagram.
The Main Coordinator

The Main Coordinator is the brain of AdviseAid. It is the central point of the program, and it decides how everything else interacts and functions.

The GUI Manager

In AdviseAid, most of the GUI fits under a managing class called the MainWindow. This class is responsible for all of the GUI components contained in the application. Visualizations, main window tabs, and other components need to work with the GUI manager to get added to the display.

Plugin System

AdviseAid has a fairly powerful and easy to use plugin system. This system is discussed in more detail later. AdviseAid makes it easy to add and remove plugins during the execution of the program, and also to modify the plugin list outside of AdviseAid environment altogether.

Help System

Another significant component is the help system. The current help system in AdviseAid uses our own XML-based system with local web pages, rather than connecting to the Internet. The help system is designed in such a way that it is easy to extend, or to replace it with a completely different system if needed.

Event Coordinator

The event coordinator is essentially the heart beat of the program, and the central component of the data pipeline. It takes data from its original source, processes it, and
prepares it for display by the various visualizations. The event coordinator is responsible for determining when more data needs to be read in from its source, when filtering should take place, and when a visualization should be updated and redrawn.

Data Sources

Data originates in data sources. AdviseAid can already access a large assortment of data sources, and the framework is designed in such a way that any new type of data could be accessed by simply creating a new input reader plugin.

Currently, there are input readers that can access specific databases, like MySQL, as well as one that goes through Hibernate, and can be configured to access just about any type of database in existence. Additionally, AdviseAid can read network traffic dump files, which contain network traffic. These files are extremely important and useful when doing network-security-related analysis. AdviseAid can also parse various other files, including things like comma separated value (CSV) files, and a variety of log files, like Snort and Dragon logs, Windows event logs, and email server logs.

The need to support multiple data sources and allow for the easy addition of additional data sources is a design consideration for the research community. This is helpful because as research advances, the source and density of data changes. Additionally, we have found the need to correlate multiple data sources. The use of relational databases greatly simplifies this correlation and allows the research to focus on the more pertinent research tasks.
**Input Readers**

Each type of data source is read in by a unique input reader. An input reader has the task of reading in data from the data source in a proper order. Typically, this is done in chronological order. Input readers are supposed to read in data one item at a time, so anything that may more easily read in large groups of data needs to account for this. Input readers also need to be able to tell the earliest time in the data set and the last time in the data set. This may require running quickly through the whole file to find this information, once the data source has been configured.

**Input Manager**

The input manager acts as a multiplexer for all input readers. The rest of the program does not care about what input reader is currently active. When the event coordinator asks for data, the input manager goes to the active input reader and collect the data from it. Within the input manager is a subsystem that keeps track of input plugins, and maintains a list of available parsers.

**Data Elements**

Data elements store data while it is in memory, from the time it is loaded from the data source until it is given to the visualization. In the code, this is represented by the ParameterPercentages class, and to a lesser degree, the DataParameters class. Filtering and searching can be performed on various data elements.
Data Pipeline and the Data Manager

As data moves down the data pipeline from the input source and the input manager, individual data elements are stored in a data manager component. The data remains there as it waits to be processed by filtering modules, visualizations, and other similar components.

Visualization Manager

Like the input manager, there is a visualization manager that is responsible for dealing with all of the visualizations. Unlike input readers, there can be multiple visualizations active at one time. The visualization manager makes sure that each visualization has the newest data and also has a subsystem that keeps track of the available visualization plugins. The visualization manager also plays an important role in inter-visualization communication, which allows visualizations to request things from other visualizations. For example, if a user selects some data in one visualization, that visualization can request the others to highlight the data as well.

Visualization

Visualizations are the central component to AdviseAid. Visualizations have the responsibility to draw information on the screen or other output destination in a meaningful way. Visualizations receive the data from the rest of the program through the visualization manager and decide how to display the data.

Usually, visualizations have mappings from the data to certain visual components that can be changed around. For instance, in a bar graph visualization, the bars might each represent a certain IP address, by default. The user could change this to represent a
different port or a different email address. These mappings are dynamically adjustable by the user, allowing for easy exploration of the data.

Also, visualizations have display parameters, which tell the visualization how the information should be drawn, for example, in a bar graph visualization, something like the background color or the maximum height of the chart.

Output Destinations

A visualization can send data to just about anywhere. So far, all existing visualizations send data to the screen. This is usually done with OpenGL, but a few visualizations use Swing, Java’s windowing toolkit and Java2D. Visualizations, however, are not limited to sending data to only sending data to the screen. A visualization could instead, print to a printer or write to a file. For instance, it is possible to create a visualization that would summarize all of the data in the data set, and generating a PDF report for the end user.

Input Processing

Datasets are an important component of AdviseAid. In the end, they are what the user wants to view, and it is AdviseAid’s job to display the raw data in a given data set in a way that allows for a user to quickly and easily extract additional information of value from it.

Initially, AdviseAid was solely focused on network security, and so it could only read network traffic data files. Over the years, though, it has expanded its scope. It began to incorporate input readers for other types of security related applications, like Snort and Dragon log files. More recently, it has begun to see a bigger need to read in
virtually any type of data, and so more general input parsers have been required, like CSV files. Additionally, over the years, it has become necessary to read from data sources beyond just flat files, allowing AdviseAid to connect directly to MySQL databases, and to virtually any form of database through Hibernate. Input parsers have become a key component of the plugin system. In this section, we will look at the various input parsers that have been created over the last several years.

Databases as Input Parsers

As AdviseAid has grown, so has its need to read from databases, specifically MySQL, instead of reading from a flat file. Up until this point, all data sets had come from flat files in a very specific format, and because of this, a number of changes were needed to allow the user to specify all of the required information to connect to a database.

To remedy this, we refactored the code to a new design that allowed for more control over a parser’s configuration. Once this was accomplished, the MySQL parser was fairly straightforward. The connection to the database was then made using JDBC.

One of the key problems visualization developers face is that much of their time is devoted to writing code to handle a certain data source. Enabling database connectivity in AdviseAid reduces this problem by allowing access to a large category of data storage previously inaccessible in AdviseAid. One of the research challenges that arises when using databases as a data source is in finding an effective way to turn database relations and their contents into in-memory data that the visualization can handle, without requiring the developers to string together complex SQL commands. In order to be
effective, this needs to be done in a way that simplifies the work of the visualization developer as much as possible.

Once AdviseAid makes a connection to the database, it needed to know what columns and what tables to query to get the information it needed, and additionally, it needed to know where to put the data. The user essentially needed to provide a way to map tables and columns in a database to attributes of the data within AdviseAid. To accomplish this in an easy manner, we developed a file format that was easy to parse and that would specify the names of the tables and columns to look up in the database, and what each column represented, as well as a flag to indicate which column in the database represented the time of the event, or other sequential ordering of the data. SQL mapping files can be created quickly using the Database Mapper tool discussed later in this chapter. An example of a file in this format is shown the snippet below.

```
# Snort Table
TABLE:Snort=Snort
TIME:Time
COLUMN:Generator=SnortGenerator
COLUMN:EventID=SnortEventID

# Dragon Table
TABLE:Dragon=Dragon
TIME:Time
COLUMN:Event=DragonEvent
COLUMN:IPRemote=IPRemote
COLUMN:PortLocal=PortLocal
COLUMN:Direction=DragonDirection
COLUMN:Flags=DragonFlags
COLUMN:Protocol=DragonProtocol
COLUMN:Message=Message

# SQL Query Log
TABLE:SQLQuery=SQLQuery
TIME:Time
COLUMN:ID=SQLQueryID
COLUMN:Command=SQLQueryCommand
COLUMN:Argument=SQLQueryArgument
```

Once we had a functioning parser for MySQL databases, we began to see a need to connect to other types of databases. For the most part, connecting to different forms of
SQL databases is very similar, especially in the case of AdviseAid, wherein we are only using the most common kinds of SQL statements that virtually all SQL databases have implemented. The only real significant difference is the fact that connection URLs vary significantly for different SQL database systems. To deal with this problem, we devised a simple method of specifying the needed information to make a connection to a database.

Our system involved creating configuration files that would specify a URL with the required parameters marked with ‘{‘ and ‘}’. Additional lines could be specified afterwards to indicate properties about that parameter, for example whether the data should be hidden, whether it should be password protected, or if the file should have any kind of input validation, such as requiring the input to be a number for a port or something like that. For example, for the MySQL example listed before, the file would contain one line that looked like this:

```
jdbc:mysql://{Database Host}:{Port}/{Database Name}?user={User Name}&password={Password}
{Password}=Hidden
```

These database connection files are all placed in a single directory and given a reasonable name that describes the connection type, so AdviseAid can quickly and easily look up the available options, and place them in a drop-down list. When a user chooses one of the options, AdviseAid builds a GUI to represent the connection type, complete with naming the fields after the names in the file, and setting correct options, for example, using a JPasswordField for hidden attributes, instead of a text file, and the type of data validation specified in the connection file would be applied to the GUI, as well.
The current version, however, still requires the user to gather the necessary DLL and JAR files to communicate with the database. This process is still fairly simple, and in the future, this could be automated as a plugin. The required DLLs for MySQL and PostgreSQL databases are already provided.

**Hibernate Input Parser**

At the recommendation of some experts in the security and visualization fields, we also added support for using the Hibernate framework in AdviseAid. Hibernate is a system designed to treat items in a database and objects in a program’s memory as one in the same. This is not the goal of AdviseAid, but Hibernate does allow connection to virtually any form of database using the same format.

In addition to the basic work of creating a new parser for Hibernate, reading data through Hibernate also requires a significant amount of configuration. Hibernate requires a series of complex XML based documents to tell it how to turn each table and column in the database into a Java object, and additional files to tell it how to operate Hibernate itself. Creating a good configuration requires a lot of work, but typically, Hibernate is configured once by an expert or developer. It rarely, if ever, needs to be changed, and the end user never needs to worry about it. So, for most typical uses of Hibernate, a long configuration process is not unreasonable.

Unfortunately, this did not work well in the case of AdviseAid, as the user needs to be able to set up Hibernate to connect to their own database. To resolve this issue, we created a subcomponent of AdviseAid that allows the user to manipulate a GUI to indicate how the various tables and columns map to objects in Java, and then save a
configuration that Hibernate can handle. Because Hibernate is so strict about what configuration files need to exist, as well as where those files should be located, this component had to be able to communicate well with AdviseAid to save the entire generated configuration in the correct locations.

To do this, we extended the Database Mapper tool that we created for MySQL databases. The details of this component are discussed later in this chapter. Here, we only discuss the implementation of saving a Hibernate configuration. Normally, the Database Mapper tool is used to indicate which Java object each table is to be converted into, and which instance variable the various columns represent. However, we configured our system such that when the user chooses to save a Hibernate configuration, a dialog box opens up that allows the user to choose a name and other properties. The dialog, however, does not allow the user to choose the location of the saved files. This is because Hibernate is often very tricky to get set up properly, and the Hibernate libraries are looking for the configuration files in a very specific location. Preventing the user from choosing an alternative location saves a great deal of problems for the user later on. If the user is using the standalone version of the Database Mapper, the user must choose a file location instead, because in the built-in version, AdviseAid is able to indicate where the correct directory is located.

*Comma-Separated Values and Delimiter Separated Values Parsers*

AdviseAid also provides an input parser to read comma-separated values (CSV) files. CSV files contain tabular data, wherein rows are placed on separate lines and
columns are separated by commas. It is an extremely simple format that a wide variety of applications know how to work with, including Microsoft Excel. Many visualizations make use of this format because of its simplicity [18].

AdviseAid supports reading CSV files, and in addition, AdviseAid provides alternatives to parse tab-separated values (TSV), and a generalized delimiter-separated values (DSV) file parser, which allows columns to be separated by any delimiter, giving the user the option to specify the delimiter in the GUI.

Each of these parsers expects the first row of the file to contain headers to describe the columns in the data set, though in the future, this may be removed by allowing the user to manually name the columns in a DSV file, and reuse the column names for multiple files.

*Live Data Sets and the Network Capture Server*

Many data sources can have new data added to them, even while AdviseAid is reading from the file, because AdviseAid typically does not lock the data source. These data sources are considered live data sets, because they are constantly changing to reflect newer data. Being able to read live data sets as new information is continually added is an important and powerful feature we implemented in AdviseAid. Many visualization techniques indicate that this feature is extremely beneficial [2, 16, 39].

The original design of the input parser system required the parser to provide the start and end time of the data set just after being configured. This information was then displayed by AdviseAid. Additionally, as data was being read from the data source, the parser was expected to indicate whether more data remained in the data set. With static
data sets, once the data source no longer contained additional data, reading the data set terminated. With its newly implemented ability to read live data sets, AdviseAid will continue to periodically request more information from the data source, and also frequently asks for updated start and end times of the data set. The user can specify which mode to run in from the main menu.

As none of the input parsers in existence were using live data sets, we created the network capture server and client, an input parser that reads live data sets. This allowed for the installation of various network capture clients across the network, which could collect network traffic on the hosts on which they reside. Once a certain amount of time has passed, or a certain number of packets have been collected, the collected data is sent across to a server program that assembles all of the traffic in one.

The client program was implemented as only a command line application, and was relatively simple. Various command line arguments configure the client, tell it how often to send information to the server, and provide the IP address of the server, as well.

One of the problems this approach faces is that depending on how the client is configured, the client could generate a fair amount of network traffic on its own. This was solved by only sending data across the network in groups, and data from the client to the server on the active port were filtered. This solution has one significant limitation, however. If interesting traffic is sent from the client to the server, this information would be filtered by the client, and the server would never see it. Because of this, filtering can be turned off on the client application. The network capture client uses the pcap libraries to collect information.
The network capture server was implemented as an input parser for AdviseAid, giving AdviseAid direct access to the collected results. The server had a thread waiting for new connections to be made, and another thread going through the open sockets and checking for additional data, pausing briefly if nothing was found on any of the sockets.

**Database Mapper**

In our initial work to connect to a MySQL database, one of our major tasks was to come up with a system for mapping columns and tables in the database to our classes and data structures. As mentioned earlier, we created a file format that specified all of the information needed to map the tables and columns in a database to features of the data set in AdviseAid. Creating these files by hand, however, is a time consuming process, and is error prone. As we began to need configuration files for Hibernate, the process became even more complex.

To deal with the issue of creating MySQL mapping files, as well as Hibernate configurations, we created a subprogram of AdviseAid called the Database Mapper. This utility program is designed to streamline the process of making MySQL mapping files as well as Hibernate configurations for use in AdviseAid. The main window for the Database Mapper is shown in Figure 1.
Figure 1. Database Mapper Utility screenshot. The database mapper utility allows a user to configure how AdviseAid uses the information in both MySQL databases, and database access through Hibernate.

The first step in creating a valid configuration is to generate the information about the database you are trying to connect to. Using the Database Mapper, tables and columns can be added and removed in the leftmost panel manually, or alternatively, an auto-load feature will connect to certain types of databases and query to find the names of the tables and columns in the specified database and build the database information for the user.

The work flow moves across the window from left to right. Once the database structure is set up in the left panel, the user can begin to map certain properties to each table in the database by choosing a type, which lets the AdviseAid know what type of
object to instantiate to represent the table, as well as a time column for each table. Once tables are configured, the user can map individual columns to object attributes in the right panel, choosing from a list of the possible options.

Once the configuration is complete, the user can save it to either a Hibernate configuration or a database mapping file; it is then loaded in AdviseAid in the Hibernate input parser or MySQL input parser plugins, respectively. Also, the user can load configuration files later on to modify them using the Database Mapper.

Data Processing

Once data is loaded into the system, the user will likely want to be able to do a number of different things with the data before it is given to the visualizations. The three most common features that we discovered were needed in a visualization, specifically in computer security visualizations were filtering [20,39], searching [24], and data mapping [11, 17, 18]. Each of these features has been implemented in AdviseAid.

Filtering

By far, the most commonly required data processing feature is the ability to filter out parts of it. This is useful because sometimes, parts of the data are not of interest, and if they can be removed, the user can more quickly analyze the significant parts of the data.

For the most part, filtering in AdviseAid is done using soft filters, meaning that the data is still collected from the input source and stored, but that visualizations never receive it until the filter is removed. This allows visualizations to immediately be updated with all data, in the event that a filter is removed. However, it maintains
overhead in that this data must be constantly skipped over by the filter parsers. Additionally, since this data remains in memory, it consumes resources.

The visualization designer does not need to worry about filtering on their own because the AdviseAid framework provides it for the visualizations. If the filtering system is not powerful enough, visualizations may still add in an extra layer of filtering if desired. Additionally, the filtering system can be accessed through the main GUI, but each visualization can update the active filters programmatically, meaning that visualizations can dynamically filter things, based on user interaction. For instance, the Connection-Based Focal Ring visualization allows users to select packets in the dataset by right-clicking, and choosing Filter from the popup menu.

The filtering GUI allows users to choose from the various attributes of the data set and choose ranges of values to be filtered out for each attribute. Multiple filters can be added at the same time, giving the user the ability to filter out virtually any of the data desired.

**Searching**

Another addition we made to AdviseAid was the ability to search through the data set. Far fewer visualization designers have requested the need to have the ability to search the data, though they exist [24]. AdviseAid’s search mechanism allows a user to specify a part of the data set in a manner similar to the filtering interface, but rather than removing the data from the visualization, it displays the details of the data within a table, and highlights the data in each of the active visualizations. This allows the user to quickly find both the details of useful information, as well as where the information is
being displayed in the various visualizations. Not all visualizations have implemented highlighting, so as a result, the information cannot be marked in those visualizations.

**Data Mapping**

One of the key features of data processing that AdviseAid provides that very few other systems allows is that of data mapping. Data mapping allows the user to manually associate various data parameters with visual attributes to alter what the visualization is displaying. For example, a user could choose either the source or the destination IP address to be used for drawing computers in a topological map. AdviseAid allows a user to apply any part of the data set to any of the available visual attributes.

Currently, each visualization supplied a list of the available visual attributes that can be mapped to, along with the data parameters that can be mapped to the visualization. In the future though, the input handler should provide a list of possible data attributes, along with metadata such as the scale of measurement (nominal, ordinal, interval, or ratio) and the data type (string, integer, decimal, etc.) and allow the visualization to determine which fields of the data can be mapped to a particular visual attribute.

AdviseAid’s data mapping tools permit a variety of data to be displayed to the visualization, or a data set to be analyzed from a variety of mappings to help illustrate different interesting pieces of information within the same data set. An example of this is the Triangle visualization, wherein mapping to the transport layer protocol reveals things such as port scans, while mapping to the packet size can reveal file transfers.
The mappings GUI, shown in Figure 2, allows users to select a running visualization and choose from the various parameters of the data the ones that they are interested in to map to the visual attributes that the visualization supplies.

Figure 2. The mappings GUI. The mappings GUI allows users to apply various data parameters to a selected visualization’s visual attributes, allowing the user to customize what data the visualization is displaying.

**Visualization Framework**

AdviseAid provides an easy to use framework that enables visualization developers to efficiently produce new techniques. The platform provides a simple and powerful API for adding in a new visualization, and additionally provides a small
collection of tools that many visualizations may find useful. The visualization tools and framework are discussed in this section.

*EasyVisualizationTemplate*

In AdviseAid, it is important to allow visualization developers to be able to work with a simple interface to create visualizations. However, the interface needs to be extendible so that more powerful visualizations can still be created. While the visualization interface in AdviseAid has grown over time, visualization developers can take advantage of the new EasyVisualizationTemplate class which we developed, to make the most of their time. This template allows developers to quickly implement the core components of their visualization, such as rendering and updating the data model for the visualization, along with other common features like providing a name for the visualization, data mappings, and saving and loading the state of the visualization. Once these core components of the visualization are taken care of, additional features of the Visualization class can be exposed by overriding default implementations.

*Geographical Maps: Google Maps in VisAlert*

A number of visualization techniques attempt to correlate events within a data set to a geographical location. This is useful for a variety of reasons. For example, it can be used to determine where on a university campus a particular problematic IP address is located, to allow the system administrator to physically travel to the location. It can also be used to determine a geographical location around the globe from which an attack is coming.
Through our work, AdviseAid provides users with the basic ability to acquire a map image using the Google Maps API. To achieve the desired result, we had to take a set of geolocations in the form of a latitude and a longitude, and determine a central latitude and longitude and a radius to obtain a rectangular map. The process is complicated significantly because most map APIs, including Google Maps, works in a rectangular coordinate system, even though in large scale maps, these are overlaid on a spherical surface. This problem was solved by applying the arc-length formula to the most distant points in the geolocation data to obtain the correct radius. This module was funded by IntelliVis, and is currently only usable from within their visualization, VisAlert. The framework code, however, could easily be extended to include other visualizations. An example screenshot of the geographical mapping in use is shown in Figure 3.
Figure 3. Geographical mapping in VisAlert. This illustrates AdviseAid's ability to acquire geographical maps and perform geolocation. It is currently only used in IntelliVis's VisAlert visualization.

**Topological Maps: Topology Forge**

Topological maps are maps showing various nodes in a data set along with their associations given by edges or lines. These maps can represent a broad variety of data, such as a network and the interconnections between computers. Many visualization techniques utilize topological maps in various forms. IntelliVis's VisAlert visualization, which is a part of AdviseAid, is an ideal example of how these maps can be used.
Because of the need for topological maps, both within existing visualizations in AdviseAid, and in computer security visualization in general, there is a need to provide a good mechanism for creating topological maps for use in AdviseAid. The topology shown in Figure 4 was created for VisAlert by hand using Microsoft Paint. The process was tedious, and did not allow for reusability, and because it produced a single image, an additional file was required to provide information about the actual location of nodes within the topology.

![Figure 4. An example topological map. Maps like this are used in VisAlert and many other computer security visualizations.](image)

To reduce the amount of time required to create a topology, we created another utility tool called the Topology Forge to quickly make topologies for use in VisAlert. This involved identifying what capabilities would be required by such a tool, identification of the methodology for linking the technique with AdviseAid proper, and design of its architecture. The Topology Forge is a powerful topology editor that includes
many features that make it easy to design virtually any type of topology. The Topology Forge can run as a standalone application, or it can be launched from inside of AdviseAid. The main window of the Topology Forge consists of an open canvas area, a properties panel on the left side, and a toolbar along the top of the window, as shown in Figure 5.

![Topology Forge Window](image)

**Figure 5.** A screenshot of the Topology Forge. This utility program is designed to make it easy to generate topological maps for use in VisAlert and other visualizations.

The Topology Forge provides users with the needed tools to create virtually any type of topology. Node types can be created with any image, and each node type can be given its own set of attributes. A topology can have any number of node types. Nodes
can be placed onto the topology individually, as well as in grids, rows, or circles, to make the placement of large numbers of nodes easy. Edges can also be drawn to connect nodes within the topology. The topologies generated with this tool can be loaded directly into AdviseAid and used by the visualizations. This tool means that topologies can be created in minutes instead of hours.

Visualizations

AdviseAid has more novel visualization techniques than any other platform that we know of. There are other visualization platforms that have a comparable number, but these rely heavily on standard plots such as pie charts and scatter plots [10, 38]. This section overviews many of the visualization techniques that have been developed to date. Each of these visualizations represents a significant amount of time in design and development.

VisAlert

VisAlert represents the most sophisticated, powerful, and full-featured visualization created with the AdviseAid framework to-date. VisAlert is the property of IntelliVis, a company in Salt Lake City, Utah. It is the prime example of what can be done with the AdviseAid platform, and because of the requirements of this visualization, the AdviseAid framework has grown to become what it is today.

VisAlert is a novel visualization correlating events to the location in which they occur and the time at which they occur. Various events or alerts are placed around the outside of a circle. The alerts can be grouped, and groups can be organized hierarchically around the outside of the circle. On the inside of the circle is a map that indicates where
the alert took place. VisAlert allows the user to use both topological maps and geographical maps. Topological maps can be generated by hand, or more recently, created in the Topology Forge tool. Geographical maps can be pulled from Google’s map API. In the plot, time is represented as concentric circles of the alert rings. The most recent information occurs at the inside edge of the alert ring, and as time passes, the alerts move outwards and eventually disappear off of the chart.

Virtually everything in VisAlert is configurable, which is very useful but also a bit of a liability, because the user needs to wade through quite a bit of configuration work. Alerts can be mapped to virtually any aspect of the data, giving the visualization the ability to see a wide variety of useful information. This version of VisAlert is designed specifically for computer and network security, and as such, it can detect a wide variety of interesting security events. We have seen the visualization detect both simple things, such as port scans, and also more complex attacks such as a person illegally gaining access to the database through a web server and retrieving all of the information from the database, including the personal information contained in it.

VisAlert has been developed well beyond the prototyping level to a production level. It takes advantage of some features of AdviseAid that most other visualizations have not, such as branding. IntelliVis distributes the entire AdviseAid platform to
Figure 6. VisAlert screenshot. VisAlert correlates events or alerts with the time they occurred at and their location on a geographical or topological map. Alerts are arranged in groups around the outside of the circle, while the location of the event is plotted within the circle. Time is represented as a set of concentric rings, with the most recent time located at the inner edge of the rings, and the oldest at the outside of the rings.

their potential clients with a customized GUI containing only the features that VisAlert needs, using text labels and images that describe VisAlert, rather than the whole AdviseAid platform.
Triangle Visualization

The Triangle visualization is a novel visualization technique based on a ternary plot. The technique was the first sophisticated technique to be built almost entirely after the core features of AdviseAid were implemented, while the majority of other visualizations were developed concurrently with AdviseAid. Because of this, the technique is discussed in a separate chapter of this thesis as a case study of the AdviseAid platform.

The concept and design of the Triangle visualization were my own novel ideas. The inspiration for the concept came from the United States Department of Agriculture’s Soil Texture Triangle, which is used by soil scientists to classify different types of soil. The Triangle visualization focuses on detecting anomalous behavior, displaying a particular attribute of the data within a triangular plot. The data must fall into one of three categories, though the three categories may be two specific categories and a third catch-all category. Each computer is represented as a single node that can change positions based on what percentage of each of the three categories it is utilizing. In most cases, typical behavior for a set of nodes will show the nodes relatively stationary, with most in the same general area. Anomalous behavior is often easily detectable because a node will be in constant motion or in a region of the ternary plot that no other node is currently in.

The details of the Triangle visualization are discussed in the Case Study chapter of this thesis. A screenshot of the Triangle visualization is shown in Figure 7.
Figure 7. Triangle visualization screenshot. The Triangle visualization is a novel visualization technique that can be used for anomaly detection by categorizing data for a particular node into one of three categories.

*Pie Chart (Advanced)*

The Pie Chart visualization, not to be confused with our implementation of a simple pie chart, is a novel technique we developed to extend the features of a basic pie chart. Multiple aspects of the data set, such as source and destination IP addresses and ports, and protocols, are plotted within a grid of pie charts. The history of the data set is plotted on concentric pie charts, giving the viewer an understanding of how the current state differs from previous states. Figure 8 shows a screenshot of the advanced pie chart visualization.
Figure 8. Pie Chart (advanced) screenshot. The Pie Chart visualization displays a wide variety of attributes in the data set and displays them in multiple pie charts. The history of the data set is displayed by drawing concentric pie charts.

The visualization allows the user to know what percent of the data set meets a certain criterion, and how the data set is changing over time with respect to that criterion. Port scans are also extremely easy to detect in the data set, as port scans produce pie charts with a large number of slices. The visualization, however, is a good example of why prototyping is beneficial, because in this particular implementation, the complexity of the display ruins any benefit that might be gained over a standard pie chart.

**ForceVis**

ForceVis is another novel visualization technique that we created. ForceVis keeps track of a set of nodes and edges connecting them, which represent different IP addresses and connections between them in the default mapping. The shape of a node changes based on how many connections are being made to the node, and the thickness and color of nodes are dynamic, but it is initially mapped to the amount of
communication between the two IP addresses. A screenshot of the ForceVis visualization is shown in Figure 9.

Nodes within the graph are drawn to the center of the screen via a default spring force to keep them all on-screen, and are repelled from each other by spring forces so that no two nodes will end up overlapping each other. Nodes that are connected together will also exert forces on each other to draw them together.

The idea behind ForceVis is that normal network traffic results in a fairly stable graph. Any sudden changes in the state of the network result in a sudden movement by the nodes.

Figure 9. ForceVis screenshot. This visualization is designed for anomaly detection using a force directed graph as the underlying structure. With the default mapping, each node represents a different IP address, while edges represent connections between multiple computers.
ForceVis has proven to be fairly useful, but the current implementation does not scale well to large networks, both in terms of performance (optimization is needed) and clarity of the display.

*Simple Pie Chart and Simple Bar Chart*

AdviseAid includes a large and growing number of visualizations available in it. While each of these visualization techniques are interesting and useful, there are some things that can best be seen with a simple graph like a bar graph, or a pie chart. These simple charts show a dataset in an extremely simple way, which has a great deal of value by itself. They are well known data display forms, and so virtually anyone can quickly grasp the content.

While they may not show everything that a more advanced visualization can show, and they may not be as flexible or powerful, simple charts still have value when used well. Additionally, as new visualizations are developed, it is important to be able to compare them to some of the tried and true methods of displaying data, such as these simple charts.

The simple pie chart visualization simply takes each piece of incoming data and assigns it to a category, based on which data attribute it is currently mapped to. For example, in a network traffic data set, each unique source IP address could be assigned a separate category. Each category would then contain all of the network packets that were sent from a particular IP address. The pie chart displays each category as a sector of a complete circle, wherein the amount of the circle the sector contains is proportional to the number of data elements in that category. Additionally, sectors that are large enough will
have a text label inside of them indicating what percent of the whole the category is. A screenshot of the simple pie chart is shown in Figure 10.

To allow the pie chart to be a bit more versatile, we allow users to hide particular categories from the display. This comes in handy when one category dominates the dataset, but that category is not particularly interesting. A user can click on it and hide it, and the rest of the data will expand to fill in the entire circle. This allows for a simple but effective form of filtering. Figure 11 shows the same data as Figure 10, with the two largest categories filtered out.

Figure 10. Simple Pie Chart visualization screenshot.
The simple bar chart is quite similar to the pie chart in its design and functionality. In fact, when it came time to design the two visualizations, we intentionally designed a common core that they could both use.

The simple bar chart shows the data categories as a series of rectangles whose heights are determined by the number of data that fits in the given category. Bar graphs function differently than pie charts in this aspect. Each group is drawn not as a percentage of the whole, but rather in proportion to the total amount of data in the given category.

Neither of these graphs removes old data from the categories, so using either of these methods with live data sets would become less meaningful after a while, as even a
significant change in the events would not alter the graph much. A chart like this is much more volatile at the beginning when there is little other data to compare it to.

In both the pie chart and the bar graph, when a user determines that a particular category may be of interest or otherwise wants to perform drilldown on a particular category, the user can right click on the category and choose a menu option to display the category’s contents. This opens up a tab in the Packet Contents tab on the main AdviseAid window showing all of the details of the information collected in the given category. The display used for drilldown is shown in Figure 12.

Additionally, because the main window is a fixed size, the user can open up an expanded view that puts the data in a completely separate window, for better viewing.

![Figure 12. The drilldown display. This display is used in both the Simple Pie Chart and Simple Bar Chart visualizations, and can be utilized by other visualization techniques.](image)
Other Components of AdviseAid

Plugin System

One of the requirements that AdviseAid needed was a plugin system. There are many companies that use AdviseAid, and each one has various components they want to be able to use, while not having the user interface cluttered with components that other companies or projects have funded. In addition, sometimes, these same companies have paid a large amount of money into a visualization technique or other components, and they do not wish to share it with other companies at no cost.

Of course, one simple approach is to develop separate branches of the software for each company, and give each one only the components that are available to them and that they wish to have, but this has its own limitations. First, such a strategy creates a large problem for developers who must fix bugs in multiple versions of the code. Additionally, it would be nice if individual users would be able to enable and disable various visualizations and components, rather than just at a company level. The visualization techniques that are suitable for one task do not necessarily work well for another [40], and so different people will likely need a customized set of tools they can modify as needed.

An ideal approach to this problem is a plugin system, which modularizes components of the program as a whole. These components can then be “plugged in” to the core framework of the software, enabling users to use them, or “unplugged” to remove them from the display. Because plugins are independent software units, they could be marketed and sold separately, though this is currently not needed in AdviseAid.
While a variety of plugin systems exist, none met the requirements that we needed for AdviseAid. Our system needed to be easy to work with, needed to be Java-based, and had to have a simple interface, so that both developers of the AdviseAid core, as well as third party visualization developers could quickly and easily build plugins.

We created our own simple system of plugins, which is built directly on top of the JAR (Java ARchive) utility, a key part of the Java platform. Java’s JAR system is designed as a simple method of packaging code. It consists of a directory of compiled `.class` files, compressed with the ZIP compression scheme, along with an optional (but recommended) manifest file, which specifies which classes are available in the JAR file, along with other options, such as the “main” class, which is the entry point to an executable JAR, if one exists.

The information contained in a JAR manifest is not very useful to AdviseAid. Instead, it is important to know which classes in a JAR file are expected to be loaded as plugins. Our approach was to add an additional file to the top level directory of a JAR file called `load.txt`, which simply lists the classes in the JAR that are to be loaded as plugins. An example load file for the VisAlert visualization might look like this:

```
adviseaid.visualization.visalert.VisAlert
```

This system easily and effectively accomplishes a number of our goals. First, it is easy to create a plugin. A developer takes the compiled class files and their directory structure, and creates a ZIP file from it, often simply by using the tools that exist in the operating system. Then, a text file, `load.txt` is created, and a single line is added to it, listing which classes are the plugins, and added to the ZIP archive. Because JAR
archives use the same compression as ZIP, the developer just needs to change the extension of the file from .zip to .jar.

Additionally, because Java uses JAR files so heavily, there is already an existing framework for opening up JAR files and using a class loader to prepare the contained classes, which made it simple for us to develop. In fact, our code for doing this is less than 100 lines of code.

Plugin Manager

Along with our plugin system came the need to have a system to manage plugins. This includes the ability to add and remove plugins, as well as see what plugins are currently loaded, and ideally, some basic information about the plugin, as well as the ability to update a plugin or find additional plugins.

Our current system does not contain all of these features. We do not yet have the ability to update plugins automatically, nor do we have anything set up in the way of finding additional plugins. The basic functionality, however, is in place.

Part of the interface for plugins requires them to specify a number of things, including short and long descriptions, a name, a version number, and a set of icons that can be used in the display to represent the plugin. Both input plugins and visualization plugins are assigned default values, so the plugin developer does not necessarily need to deal with this, though it is typically done.

The plugin manager shows a hierarchical display of the installed plugins, displaying their icon and name. When a given plugin is selected, the plugin description box is filled in with the plugin’s information, specified by the developer. This
information allows for HTML, so formatting and styling information can be added in to the description, as well as hyperlinks. Figure 13 shows the plugin manager window.

![Plugin Manager](image)

Figure 13. The Plugin Manager interface. Plugins are categorized in the tree hierarchy on the left, with detailed information about the plugin on the right, along with options for updating or uninstalling the plugin, as well as installing new plugins.

From the plugin manager, plugins can be uninstalled, as well as installed, which allows the user to browse their file system for a .jar file that contains the desired plugin. Once a plugin is added to AdviseAid, it does not need to be added again, unless the user removes it.

Web Demo Version

As AdviseAid has grown, so have the opportunities for giving demonstrations of it. In the past, this has been done by packaging up a version of AdviseAid with the latest
stable code and delivering it to the demonstrator, who places it on his or her computer
and shows it to potential clients.

IntelliVis also wanted to be able to do demonstrations in a web-based form so that
they could simply connect to our up-to-date server, log in, and see a fully functional
version of AdviseAid that they could display to their potential customers.

Beyond just making it easy to give demos, having a web-based version of
AdviseAid provides a number of interesting advantages. First, users do not need to
install anything. Second, updates can be done quickly and easily on the server end, and
the client knows that they are always running the latest version of the system. Third, a
user can access it from any computer that has Internet access, so they can use it from
home, work, or on a business trip.

The web-based version of AdviseAid required a number of changes to function as
a Java Applet instead of a Java Application. The biggest and only significant challenge
we faced in this conversion was the ability to load the resources that were needed for the
program. This included everything from static resources, such as icons and branding
information, datasets, and external libraries.

Loading data sets in the applet version was complicated for two reasons. First, in
the desktop application, choosing a dataset to load required using a JFileChooser, a Java
base file dialog that allows browsing a directory structure. Because of security
restrictions placed on Java applets, it was impossible to allow the applet version to access
a file chooser on the server computer. Unsigned applets are fairly restricted in what they
can do. They are not allowed to create a network connection to any machine but the
server machine they were launched from, and they are not given access to the local machine’s file system. To allow the user to load datasets, we created a file server program that ran on the server at the same time as the web server. The client applet could then open a normal TCP connection to the file server software to perform a number of tasks, like querying which datasets were available, and asking for the server to stream the contents of a file back through the socket so that the applet could read the dataset. In addition, the program needed to be reconfigured a bit to allow input parsers to read data from input streams in general, instead of from files specifically.

Loading the required libraries was also a challenge for an applet version of AdviseAid. Within AdviseAid, there are two types of libraries commonly used. First, there are libraries containing Java classes that need to be used. These are contained in JAR files. These are fairly easy to load in an applet, because an applet is designed to use JAR files. Consequently, these only needed to be added to the list of libraries in the applet codebase, which is a part of the HTML tag for the applet. AdviseAid also uses a number of native libraries, like the DLLs needed for OpenGL. Applets are not typically allowed access to load DLLs, because they can execute native code that could attack the computer, outside of the sandbox that a Java applet is given. For some of our native libraries, this meant that we could simply not use them in the applet version, and some of the minor features of the program are disabled in the web-based version, such as image and video screen capturing. Other libraries were offloaded to the file server software, and the results were streamed across to the client. One native library that we simply could not do without was the libraries for OpenGL. It would be impossible to have the server
handle the rendering of each visualization and send it to the client. Further, it was also impossible to simply load the DLL that was needed for this. This particular problem was solved by using a JNLP application alongside the applet to load the required native libraries so the applet could use it. The JNLP application launches and notifies the user the first time of what it is going to do. JNLP applications typically have fewer restrictions, because they are treated more like downloaded applications. When a JNLP application is loaded, it tells the user what it is going to be doing, and the user can block it or not.

A web-based version also presents other challenges and problems. Some of these are inherent to web-based systems, and cannot readily be solved. Others are more specific to AdviseAid, and have been dealt with. Others, still, are problems with AdviseAid itself, but the challenges have not yet been resolved.

Any web-based system has the added problem of needing to manage bandwidth issues and server load constraints. Because we are using a Java Applet, the program runs mostly on the client end, which limits the amount of calculations that are performed on the server end, and while initially requiring a larger transfer, afterward requires transfers when the client requests a new data set from the server. This has helped to reduce the amount of bandwidth and backend server computational power that is needed for AdviseAid.

In the end, though, the creators of Java recommend using JNLP over applets now, and AdviseAid would be better suited to take advantage of the benefits that come with it, rather than all of the restrictions of applets. In the future, this could be an important
change to the web demo version of AdviseAid, as well as provide a better distribution
mechanism for the program.

*Image and Video Capturing*

One of the most common things a user of a visualization needs to be able to do is get screen captures of the visualization of a particular data set. These screen captures are often still images of the visualization at a particular point in time, but occasionally, a video screen capture is required. These screen captures are often used by the developers to demonstrate how their visualization works, but they can also be used by the end user to present findings to others. Because this need is so commonplace, AdviseAid provides these features as a standard part of the framework.

A variety of screen capture tools exist on the market today. However, there are a number of reasons why these tools may not be sufficient in many visualization environments. For example, in many government fields, because of security and clearances, a screen capture tool may not be approved for use, whereas the visualization tool is. Along the same lines, even for those who are able to use a screen capture tool, it may often be easier to use a built in system for doing this, rather than running a separate program.

Going a step beyond most of the existing video capture tools, our built-in system is able to capture windows that are minimized or hidden by another window, so a user does not need to have the window in plain view to get a screenshot or video capture of their visualization.
Image and video capturing can be easily accessed by using the Capturing tab on the main window. Within this window are two tabs. The first tab allows a user to create an image or video capture of any of the visualizations that are currently open. The image capture GUI is shown in Figure 14.

A user can create an image capture by clicking on the camera icon. To start a video, the user clicks on the video camera icon. Still screenshots, as well as video captures are stored in a temporary location on the hard drive, and given default filenames that encode the visualization, date, and time of capture. Users can then view the image and video captures that have been taken, using the capture manager interface, shown in Figure 15.

Figure 14. The image and video capturing interface.
Figure 15. Video and image capture manager screenshot.

At the bottom of the window is a list of thumbnails of the images and videos. Clicking on one causes a larger version to appear in the main upper panel, along with the information about when it was created, as well as what visualization it was. The user is given the option of saving the file elsewhere on the file system or deleting the capture entirely.

**Branding**

Funding for various components of AdviseAid have come from a wide variety of sources, including private companies and the United States Air Force. A number of these companies expressed interest in having a version of AdviseAid branded or customized in a particular way to reflect their own needs. For example, some AdviseAid clients wanted
IntelliVis to be able to demonstrate an example of their VisAlert visualization method to their own clients. They wanted a version that would display their own logo image on the start screen, as well as in the title bar, and other places. Additionally, they did not want many of the GUI components to be displayed either, since it was more controls than they needed, and detracted from the demo.

We created a simple branding system that met these requirements. It was powerful enough to allow for different text strings throughout the program, different splash screen images and icons, and the ability to show or hide components of the GUI.

The system utilizes two files. The first is a file with the extension `.brand`. This file lists the bulk of the branding information, including the files to use for various resources like icons and the splash screen image, as well as a listing of whether or not to show various components of the GUI.

A brand file is loaded into the program, and the contents are dumped into a dictionary that is accessible from any part of the program as key and value pairs. As various components are being initialized, they can look up the key they want to check and get the result. For instance, as the GUI is being initialized, it can check to see if a particular component should be visible or not, and add the component, based on the value in the dictionary.

One of the required key elements in the `.brand` file is a reference to a file containing various text strings. This file is called the string file and has the extension `.str`. The string file is the second important component of branding. It is similar to the brand file, but instead of specifying global program options, it lists what strings are to be
displayed in various components of the program. This, for instance, allows a title string
to be specified, which is then placed on the title bar. The normal version of AdviseAid is
given the title “AdviseAid”, while IntelliVis’s version can be branded to say “VisAlert”
on the title bar. A large number of strings are listed in the string file.

As an example, below are the contents of the default AdviseAid brand file.

```plaintext
SPLASH_SCREEN_IMAGE=Resources/Brand/AdviseAid/SplashScreen.png
SHOW_MAIN_TAB=true
SHOW_MAPPING_TAB=true
SHOW_BEHAVIOR_TAB=true
SHOW_PACKET_CONTENTS_TAB=true
SHOW_DATA_DETAILS_TAB=true
SHOW_CAPTURING_TAB=true
SHOW_SEARCH_TAB=true
SHOW_FILE_MENU=true
SHOW_FILTER_MENU=true
SHOW_HIGHLIGHT_MENU=true
SHOW_OPTIONS_MENU=true
SHOW_HELP_MENU=true
SHOW_SELECT_INPUT_MENU_ITEM=true
SHOW_MANAGE_PLUGINS_MENU_ITEM=true
SHOW_LIVE_OR_STATIC_DATA_COMPONENT=true
SHOW_LIVE_DATA_MENU_ITEM=true
SHOW_STATIC_DATA_MENU_ITEM=true
SHOW_SAVE_MENU_ITEM=true
SHOW_SAVE_AS_MENU_ITEM=true
SHOW_OPEN_MENU_ITEM=true
SHOW_DATABASE_MAPPER_MENU_ITEM=true
SHOW_CHANGE_CAPTURE_FILE_MENU_ITEM=true
SHOW_RESTART_FILE_MENU_ITEM=true
SHOW_EXIT_MENU_ITEM=true
SHOW_MANAGE_FILTERS_MENU_ITEM=true
SHOW_CLEAR_ALL_HIGHLIGHTS_MENU_ITEM=true
SHOW_COMPUTE_INTERVAL_MENU_ITEM=true
SHOW_TIMING_VARIABLES_MENU_ITEM=true
SHOW_DISPLAY_PARAMETERS_MENU_ITEM=true
SHOW_SET_LOOK_AND_FEEL_MENU_ITEM=true
SHOW_JAVA_LAF_MENU_ITEM=true
SHOW_NATIVE_LAF_MENU_ITEM=true
SHOW_NIMBUS_LAF_MENU_ITEM=true
SHOW_ABOUT_MENU_ITEM=true
SHOW_PROGRAM_HELP_MENU_ITEM=true
SHOW_CONNECTION_OR_PACKET_MODE=true
STRING_FILE=./Resources/Brand/AdviseAid/AdviseAid.str
```
The string file allows for user defined variables, which can be reused later in other strings. For instance, many of the strings in a program may refer to the title of the program. If the title changes (for example, from “AdviseAid” to “AdviseAid 2.0”), the variable storing the title is the only place a change is needed, and every other string will be updated to reflect this.

A portion of the default AdviseAid string file is shown below.

```
$TITLE=AdviseAid
SplashScreen.WindowTitle={$TITLE}
SplashScreen.MainText=Loading {$TITLE}:
SplashScreen.InitDisplayParameters=Initializing Display Parameters
SplashScreen.BuildingVisualizations=Building Visualizations
SplashScreen.InitGUI=Initializing GUI
SplashScreen.InitEventManager=Initializing Event Manager
SplashScreen.PreparingInput=Preparing Input
MainWindow.BasicTitlebarText={$TITLE}
MainWindow.ExtendedTitlebarText={$TITLE}:
Menu.Help.ProgramHelp={$TITLE} Help
```

This simple brand system has proven to be quite effective and easy to use. In addition to fulfilling the original goals, the string file acts as an easy method for handling internationalization and localization. While this has yet to be fully implemented in AdviseAid, it is a trivial task to create another string file that contains another language such as Spanish or French. AdviseAid could just as easily load that string file instead, providing a simple mechanism for providing language translations for the application.
RESULTS AND CONTRIBUTIONS

AdviseAid has changed the way computer security visualization is approached in a number of ways. The architecture allows researchers to quickly and easily implement a new visualization technique in a way that is separate from the framework, allowing them to concentrate on the actual technique. Our framework combines a wide variety of tools and features that are important to visualizations that many other systems fail to include. AdviseAid also allows visualization designers to quickly implement prototypes to determine the viability of a technique, while simultaneously providing enough resources and computational power that visualizations can be produced at production-level. These features are discussed individually in more detail below.

Comparison of Features

AdviseAid provides many features that visualization designers are clamoring for. In terms of usable features for security visualization tools, AdviseAid performs significantly better than other visualization systems.

AdviseAid provides a wide variety of features to visualization designers that no other system provides. AdviseAid allows the user to add in custom made visualization techniques, and work with multiple visualizations simultaneously. Other systems do this to a limited level only. Unlike the other visualization systems, AdviseAid allows the user to animate a data set, and control the speed of processing. While many individual visualization techniques have either implemented this [3, 11, 32, 39], or have indicated that they want to implement it [2, 9, 12], AdviseAid is the only visualization platform that currently supports it.
With input, each of the existing visualization platforms expects a specific kind of input, while AdviseAid provides a wide variety of input sources, it allows developers to easily add their own new types in as needed. Each of the existing platforms fails to provide important mechanisms for handling data, while AdviseAid allows users to filter and search data sets, and perform data mapping. AdviseAid also provides more support to new visualization techniques by providing built in mechanisms for drilldown, as well as for generating geographical and topological maps.

**Novel Visualization Techniques**

In addition to the development of AdviseAid’s core framework, a wide variety of novel visualization techniques have been created as well. AdviseAid currently has fifteen visualization techniques, eleven of which are novel visualizations. This shows us that the AdviseAid platform allows visualization designers who have created a novel technique on paper to quickly and easily develop a working version of their design. While many of the existing visualizations have a comparable number of existing techniques, their techniques typically consist mostly of visualizations that are more or less simple graphs, such as pie charts or bar charts. AdviseAid only provides minimal support for those common graphs, but instead, is focused on more advanced visualization methods.

**Separation of Technique and Framework**

One of the key results of AdviseAid is a strong separation of the visualization technique from everything else. It is clear from the way the literature describes other new visualization techniques that the visualization creators have focused on a wide variety of supporting code beyond just the actual visualization. AdviseAid has produced a system
wherein a developer only needs to implement the rendering and processing that is specific to their visualization.

**Prototyping**

One of the key features that AdviseAid provides is the ability to easily prototype a new visualization idea. The idea of a prototyping framework is significantly different than previous approaches to security visualizations. In the past, novel visualization techniques have been built by creating a new standalone application with limited features. When the technique has proven useful, the designers have then gone back and added in the desired features.

The prototyping process of AdviseAid has proven to be extremely useful in at least two different scenarios. First, the Air Force’s Rome Labs approached us with several possible visualization designs, and asked us to implement one for them. Instead, we began implementing two of their designs. Within a short time, we had two working prototypes and were able to determine that one technique was far more useful than the other. We were able to then focus the rest of our time on the more useful visualization technique.

AdviseAid provides developers with the ability to quickly implement fully functional prototypes. The best example of this is the Triangle visualization. The Triangle visualization was designed on paper in a matter of days, and then within a week’s time, a fully functional prototype was implemented. The entire implementation process took no more than forty hours of work, leaving time for tweaking and testing the
design. The Triangle visualization is discussed in more detail as a case study at the end of this thesis.

**Production-Level Development**

Prototyping is a key component of the development of visualizations. Once a prototype is built, and a new technique is proven to be effective, many visualization developers will likely begin work on a completely separate visualization application. However, AdviseAid provides enough power and features to allow a prototype to be turned into a production-level visualization.

The strongest evidence of this is that of the VisAlert visualization. Once VisAlert was prototyped, the designers, IntelliVis, were able to provide feedback and additional requests to make the visualization more powerful. IntelliVis funded us to add many additional features to the framework, such as allowing their visualization to read from the data sources they wanted. Then, using the branding framework that we developed, the entire application was customized and branded to appear as an entirely different application, specific to their needs. IntelliVis is currently marketing their visualization technique.
FUTURE WORK

AdviseAid has become a powerful program for developing and testing visualization techniques, and it is even powerful enough to be used as a commercial framework. However, it has a number of limitations that need to be improved upon. In this section, we take a look at some of the limitations and shortfalls for AdviseAid, as well as looking at plans for the future.

Many of the current shortfalls of AdviseAid have arisen as a result of the moving target problem. As time has passed, the goals for AdviseAid have grown and changed. These changes have made AdviseAid fairly powerful, but the constantly changing requirements have created a large number of inconsistencies in AdviseAid that will need to be addressed in the future, to allow AdviseAid to grow further.

Rework of the Data Pipeline

As the data sources for AdviseAid have become more diverse and AdviseAid has expanded beyond network capture files and beyond simple security-related data to generalized data sets like CSV files, there is now a need to rework the data pipeline in its entirety. The current data pipeline is inefficient. Currently, the data is read in to two classes, one of which is the ParameterPercentages class, and the second is the DataParameters class. Originally, these classes essentially contained all of the meaningful information that the header of a network packet might contain, including source and destination IP addresses, transport layer protocol, packet size, and time of transmission. As the need for additional types of data with additional fields arose, however, new instance variables were added to the ParameterPercentages class. For
instance, when Snort data was added, three additional parameters were added for the
generator number, the event number, and the revision number. A large variety of data
types have been added, and for a while, each new type required additional instance
variables, to the point that now there are 171 instance variables for each
ParameterPercentages created, making each item in a data set nearly one kilobyte in
size, once loaded into memory. Within each ParameterPercentages object, only a
handful of fields are actually in use, creating a huge amount of wasted memory for
virtually all types of data.

Over time, the AdviseAid team has used multiple approaches to reduce this
growth as new data types have been added. First, we put data into fields that did not
necessarily match the original purpose. This meant that data was not labeled correctly.
Thus, while it was visible to the programmers, it was hidden from the end user.
Consequently, an extra amount of care was required in working with data, while giving
the benefit of preventing further expansion of the ParameterPercentages class. This
method, however, was far from ideal.

With the most recent new data types, we added in a system of extensions to the
data pipeline. The extension system allowed for new fields to be added to the storage in
the data pipeline without having to add the fields to all of the data storage throughout the
pipeline. While this extension system nearly solves the problem, it still leaves some
loose ends. Currently, each data item still has the 171 original fields in addition to the
new dynamic field storage that comes from the extension system.
To solve this problem completely, we will need to refactor all of the 171 fields into the extension system, along with any place throughout AdviseAid that references these fields. Since the data pipeline covers everything from the input readers to the visualizations, this is no small task, and will require changes to virtually every component of the system, or to deprecate parts of the system. The change, however, is one that is needed in the near future, to allow AdviseAid to better handle data sets that contain non-standardized fields, like XML files, or CSV files.

**Plugin System Improvements**

As discussed earlier, the plugin system is currently extremely simple and powerful. The system, however, still has three significant limitations that could easily be addressed to give AdviseAid plugins far more power. At the moment, all plugins must be either visualizations or input readers. Ideally, a developer should be able to make a plugin that can make virtually any change to AdviseAid, like adding content to the help system, or adding a menu item.

This would require changing the way plugins work, such that they install themselves, rather than being installed by the visualization manager or input reader manager. However, because input reader and visualization plugins are the most common, a default system should be built for these two types, as well as possibly other types.

Second, plugins should be able to specify prerequisites when they are being installed. Multiple plugins may desire to reference a common library, and so the library ought to be able to be accessed as a separate plugin that all of the others can refer to.
Finally, there needs to still be some significant improvements to the way plugins are updated. Currently, all plugins must be updated manually, which complicates the process and may prevent users from keeping up-to-date with the newest versions of various plugins. The plugin system will likely need to be extended somewhat to allow each plugin the ability to specify where new updates will come from, like a URL. Such capability could then provide the AdviseAid system with the ability to lookup what version of the plugin to update to, and what version of AdviseAid is required, as well as potentially allowing the user to choose whether to get development or beta versions of plugins, rather than just final release versions or maintenance versions.

**Improved Mapping System**

One final area of improvement that AdviseAid will need soon is a revamped mapping system. The current mapping system, while functional, is very basic. The GUI for the current system is shown in Figure 16.

In the current version, each visualization is responsible for providing a list of data parameters that the user can map from, and a list of visual attributes that each of these parameters can map to.

The current system has several limitations that need to be resolved to allow AdviseAid to continue to be a useful application and development platform in the future. First, the data source needs to be the origination point of the data parameters. The type of data being viewed is determined by the input parser and the contents of the file. Currently though, the visualization is the object providing the list of data parameters, and they do not use any information about the currently loaded data set or data parser. (It is possible
Figure 16. A screenshot of the current mapping system GUI. Each visualization provides a list of parameters from the data that they can map from, and a list of visual attributes that can be mapped to. Users can pair any data parameter provided with any visual attribute provided.

for a visualization to determine this information by working with the input parser, but the process would be complicated and long, so no visualization does this.)

Second, the current system allows a user to map any of the listed data parameters to any of the listed visual attributes, without regard to the type of data it represents. This causes serious problems with the way data is handled. For instance, consider a visualization like the Triangle visualization that averages a parameter of the data and uses it as an important part of the display. Some fields in a data set can be averaged
meaningfully, while others cannot. The packet size of multiple packets can be averaged, and can thus be used for this function in the Triangle visualization. However, the transport layer protocol of multiple packets cannot be averaged meaningfully. For instance, if the visualization considers a set of eleven packets wherein nine are TCP packets (represented with the protocol number of 6) and two are UDP packets (represented with the protocol number of 17) when the average is taken, the result is eight, which is the protocol number for the Exterior Gateway Protocol (EGP). This could lead a person to think that EGP is being used, when in fact, only TCP and UDP are being used. This form of data cannot be averaged and could, therefore, not reasonably be mapped to this particular visual component of the Triangle visualization. The data is nominal or categorical, and so this operation cannot be used for this purpose.

AdviseAid’s mapping system needs to be expanded to allow visualizations to filter which types of data from the data set and input parser can be mapped to a particular attribute of the visualization.
CONCLUSIONS

The process that AdviseAid has gone through over the last three-and-a-half years has shown us a great deal of things that may help in the field of visualization in the future, as well as give direction to where AdviseAid will go from here.

Our survey of some of the more popular security visualization techniques has given us important information about what features most visualizations need. These features include reading from files and other data sources, a close connection with other visualizations that may be running simultaneously, a component of drilldown, data filtering and searching, and a system to easily deploy the new technique to potential clients, as well as screen and video captures for reports.

The progress that has been made over the last several years has made AdviseAid a useful platform for creating and experimenting with new visualization methods. We have seen that the process of making a quick prototype for a visualization is extremely useful, because it has allowed us to quickly determine if a particular visualization method can accomplish the task it was designed for. We have seen a number of visualizations that, once implemented, were clearly not going to be able to do what we wanted of them. For example, the Time Analytic visualization was dropped early on, once we had a working basic prototype, because we could see that the Trilogy visualization was a much better choice for the task they were both designed for, and time constraints required us to complete only one. Others, still, have exceeded even our own expectations. VisAlert, for one, has turned out to be the most fully developed and powerful visualizations, and as the initial round of implementation was completed, we could then see many ways to extend
and improve the visualization for the better. Once we had prototyped several of our visualizations, we could easily see what they were lacking, and come up with a quick fix to the problem, without needing to develop a complete and individual package for the visualization. We have seen that the prototyping process that AdviseAid creates greatly helps with the development of a new visualization technique.

AdviseAid makes it easy to implement new visualizations because it allows developers to concentrate solely on their specific visualization technique. Many of the common tasks that they all share are done within the AdviseAid framework. Visualization developers do not need to worry about reading in files, connecting to databases, or filtering data, and the framework provides a large number of other tools that help in the development and testing of a new visualization. Developers have multiple options for the rendering of their visualization in OpenGL and Swing, and while it has not yet been tested, they could theoretically develop in any language that runs on the Java Platform.

The growing collection of implemented visualization techniques has reinforced the idea that visualization is very beneficial in computer and network security applications, as well as in broader fields. We see visualization methods giving users strong visual cues that an attack is occurring on their network, and helping them to understand what is happening.

In addition to the benefits that we see from AdviseAid, we also see a couple of drawbacks to using visualization techniques in general. From what we have seen, it would appear that a single visualization alone will not likely be able to identify all attacks
and threats, or even the majority of them. Visualizations often need to work in conjunction with others to give users the information they need. For example, a rule-based visualization, like VisAlert will miss previously unseen attacks that the user has not accounted for. A visualization like the Triangle visualization will not always be very accurate with previously known attacks the way VisAlert is, but it does give users the benefit of detecting anomalies. A combination of the two visualizations would produce better results than each one individually. Indeed, AdviseAid provides a great deal of support in this manner, by allowing a user to run any number of visualizations simultaneously, and by allowing users to select data in one visualization that is then marked in other visualizations.

While we can see with AdviseAid that visualization is a very useful tool in attack detection, it is unlikely that visualization will ever take over the role of many other security features, such as a firewall, intrusion detection systems, and other security tools like Snort and Dragon. These tools perform a variety of computations on the raw network traffic and take action as necessary. The goal of visualization is not to perform raw calculations, but rather, to display data in a way that the human eye can process faster than it would if the user were given a large data set in a table or text-based form. As research in all of these areas progress, we will likely see more powerful visualization methods using data created by more powerful intrusion detection systems, and more powerful security tools, rather than only visualization or computational tools dominating the field.
CASE STUDY: THE TRIANGLE VISUALIZATION

Introduction

As a demonstration of the usefulness and versatility of the AdviseAid environment, I created a novel visualization technique designed specifically for anomaly-based intrusion detection. The goal of this visualization is to demonstrate how easy it is to create a visualization using the features AdviseAid provides, as well as to produce a new visualization technique that has value on its own, and expands the current field of knowledge of using visualization for computer security purposes.

We used the Triangle visualization technique for our case study, which is an extension of a ternary plot. A ternary plot is a type of graph that plots points within a triangular region, based on what percent of a whole matches each of three separate categories, as shown in Figure 17.

While not nearly as common as other plots, such as a bar graph, or pie chart, ternary plots have been used for a number of purposes. A prime example of a ternary plot in relatively widespread use is that of the United States Department of Agriculture’s soil texture triangle [35]. The soil texture triangle is used by the USDA and soil and plant scientists to label and categorize soil measurements. When using the soil texture triangle, soil is considered to have three attributes (silt, sand, and clay), each of which are a certain percentage of the whole.

The triangle visualization plots points in a triangular region on the screen. Taking advantage of data mappings, one of the key components of the AdviseAid framework, the points plotted in the triangle visualization may represent virtually any aspect of a given
Figure 17. A basic ternary plot. Ternary plots measure three components that combine to form a whole. Based on the values along the three axes, a point can be plotted within the triangle representing the given percentages.

By default, each point represents data transferred between two computers, so two IP addresses uniquely identify a connection. A node’s position in the triangle can be mapped to a number of different attributes about the connection, and the node moves around over time, as the given attribute changes. The three attributes that we are specifically concerned with are port, protocol, and packet size, though AdviseAid gives us the ability to map to other attributes of the data set as needed.
The triangle visualization is primarily useful as an anomaly-based intrusion detection visualization, wherein the user is able to quickly and easily detect when something abnormal is happening. Anomaly-based attack detection is known to be a very powerful form of detection because it allows a system to discover attacks that are not previously known. Current behavior is compared against “normal” behavior, and any time it deviates from the known normal behavior, the user is alerted that something may be wrong.

The triangle visualization expands on the basic ternary plot in a number of ways. First, because the visualization occurs over time, the plot is animated. As time passes, nodes move around within the triangular plot area, and the user sees how the network is currently behaving, as well as how it might be changing.

Second, to make the changes more obvious, and also to show the nodes’ history, a trace line is drawn behind each node to illustrate the path it has taken.

Third, as a method of better illustrating normal behavior, as well as anomalous behavior, the triangular plot area is divided into a given number of smaller triangles called zones. All of the area within a single triangular zone represents similar data. Zones for a selected node are colored based on how much time the node has spent in the given zone. Colors are customizable, though the default is to color the zone black if the node has never spent any time in that zone, solid blue if the node has spent a great deal of time in the zone, and a shade in between if it has spent a smaller amount of time in the zone. If a node moves into a zone that it has never been in before, it will draw the user’s attention to the anomaly.
A screenshot of the completed Triangle visualization is shown in Figure 18, illustrating the basic features of the visualization.

![Figure 18. A screenshot of the Triangle visualization. This shows the basic plotting, along with traces and zones, colored in blue, for the selected node.](image)

**Previous Work**

Detecting and analyzing attacks on a computer system is a difficult task. In fact, it may be impossible to come up with a perfect solution for this problem. There are a number of complicated problems that must be addressed in order to come up with an effective solution. First, on a network, or even just a single computer system, there is a massive amount of data that needs to be processed. Within any system designed for
intrusion detection, an analysis must be able to process this data quickly, in real-time or near real-time, to be effective.

Second, on any given system, the majority of the data likely represents good or normal use of the computer network. An intrusion detection system or analysis tool must be able to effectively sort out data that represents an attack from data that represents normal use of the system. Of particular interest here, are the concepts of false positives and false negatives. A false positive in this context is when an intrusion detection system identifies something as malicious activity when it is really normal or valid. A false negative occurs when a system indicates that something is normal, or fails to mark it as malicious, when, in fact, the event was part of an attack. Ideally, a system should have both low false positive and low false negative rates. It is generally thought that having a low false negative rate is somewhat more important, because false negatives mean that attacks are going by unseen, whereas false positives means the analyst has more data to look at, but nothing malicious is getting through. Thus, a good intrusion detection system or analysis tool has low error rates, especially low false positive rates.

Thirdly, it is important for an intrusion detection system or analysis tool to be able to identify new attacks that have not been seen before. This is a critical feature of this type of system, because attackers are always coming up with new attacks and exploiting new vulnerabilities.

A variety of methods have been applied to the problem of intrusion detection. Kabiri and Gohrbani [11] point out some of the various types of techniques that have been applied in this area, including artificial intelligence, rule-based systems, data
mining, Bayesian classification, and fuzzy logic. They point out that each of these methods has advantages and disadvantages that come into play in computer security.

One method for intrusion detection or analysis that has begun growing in usefulness is that of visualization. Ball et al. [2] point out that in the communications they have had with network analysts, the analysts have indicated that they have many tools to use, most of which are text-based. They state that the analysts are requesting more visual tools to help them quickly see the state of their networks.

Another trend in the field of intrusion detection is that of anomaly detection, rather than signature-based or rule-based systems [11]. This is because signature-based and rule-based systems essentially provide a blacklist of the different activities that constitute malicious activity. When new attacks appear, these systems generally miss the attack. Anomaly detection will most likely be able to detect the attack. Several anomaly-based intrusion detection systems have been, or are being developed. Depren et al. [5] for example, designed a system with anomaly detection and misuse detection, utilizing self-organizing maps (SOMs). A number of others have also applied anomaly detection to intrusion detection [14].

By combining these two trends, we can see that an anomaly-based visualization approach to intrusion detection and analysis may prove useful. However, many of the current anomaly-based visualizations, such as AdviseAid’s ForceVis, fail to truly provide any new data beyond other existing visualization techniques, and there is still an empty hole that needs to be filled in this area. From a survey of literature on visualizations geared towards computer security, it appears that our approach of extending a ternary plot
into an anomaly-based visualization tool has not been rigorously applied. Our design is the first to attempt this that we are aware of.

In addition to the goal of designing and developing a useful new visualization technique, our other goal was to more fully test out the features and tools that AdviseAid provides. To date, all visualizations that exist in AdviseAid have one of two problems. Either they are too small to fully test AdviseAid’s usefulness, such as the simple Pie Chart and simple Bar Chart that were discussed earlier in this thesis, or they are sufficiently complex, but were developed mostly before many of the more powerful features of AdviseAid were created. The Triangle visualization is the first large scale attempt to utilize the features of the AdviseAid environment.

**Methods and Implementation**

The Triangle visualization is a prototype visualization, and as such, many of the aspects of the design and implementation were determined by this fact. For example, the triangle visualization’s rendering was done with Java’s Swing and the Graphics2D class, as opposed to OpenGL, which is used by most of the other visualizations in AdviseAid. This was done to allow for rendering that is easy to work with and looks nice, without having to do a lot of work.

**Overview**

The basic structure of the Triangle visualization is fairly straightforward, and essentially follows the Model-View-Controller design pattern. The architecture is illustrated in Figure 19.
Figure 19. Illustration of the overall architecture of the Triangle visualization. Input comes in from AdviseAid, the underlying model is updated to include the new input, and the renderer displays the changes in the user interface.

The visualization takes advantage of many of the features that AdviseAid provides, and as such, the visualization has no need to worry about input handling, other than to work with the data that AdviseAid gives to it.

The underlying data model consists of a collection of nodes, which each represent a single point plotted in the Triangle visualization. Each individual node consists of three major pieces. First, there is some matching information, used to determine if new data belongs to the node or not. The code is designed so that the matching information is an interface, which allows for developers to easily extend the visualization, but the default implementation allows for any network packet that has the same source and destination IP addresses to be considered a match as described in the introduction of this chapter. If
no match is found in the set of nodes, a new node is created, with appropriate matching information for future use.

When a new piece of data is added to a node, the node updates itself to include the new information. At the core of this process, the node updates its trilinear coordinates, which determine its location.

Additionally, a node contains a history of where it has been in the recent past. As enough time passes between updates, all of the nodes move to the next time step. The history of a node is stored as a queue, with new time steps being added to the back and the oldest time steps being removed from the front when the queue becomes filled to a given maximum capacity. Beyond the basic functionality of a queue, all time steps in the list are visible externally, so that the contents of the history can be displayed to the user on demand.

Each node also contains a zone manager, which takes the current location of the node within the plot and maps the value to a particular zone. A zone is a triangular region within the full triangular plot. Zones are a way of effectively grouping similar points together into a single bucket. As a node moves around, it spends more time in particular zones than others. In most cases, a node spends all of its time within a single zone, or a small number of zones, and when the node suddenly leaves the zone to a different zone, especially one that is drastically different from where it was before, it becomes obvious to the user visually that something abnormal has happened, and the node is no longer following normal behavior.
At each frame of the visualization, the renderer redraws the screen to display the most recent contents of the data model, which consists of the current points, their histories, and rendering of the zones of selected nodes. The rendering, along with various settings that the user can modify during run time, form the view component of the model-view-controller design.

*Trilinear Coordinates*

Trilinear coordinates are a critical component of the Triangle visualization. They determine where a node should be plotted on the ternary plot. Trilinear coordinates consist of three values \(x, y,\) and \(z\), where \(x + y + z = 1\). Trilinear coordinates differ from 3D Cartesian coordinates in that the values combine to form a whole. While this provides the ability to plot ternary points on a graph, it is also a limiting factor in the sense that any of the components can be—and must be—determined by the other two values. For example, if \(x = 0.2\) and \(y = 0.3\), then we know \(z\) must be 0.5.

This has a direct impact on the way the visualization works, and is the single largest limitation to a ternary plot, in general. For example, if each node represents the flow of communication between any two IP addresses, the trilinear coordinates, which tells us the location of a node, can represent the protocol used to communicate between the two. The \(x\)-coordinate represents the total percent of packets that use TCP. The \(y\)-coordinate represents the total percent of packets that use UDP. The \(z\)-coordinate represents the remainder of the packets, which are all of those that are using another protocol besides TCP or UDP, such as ICMP.
In some ternary plots, such as the USDA soil texture triangle, there are only three possible categories or labels, and each coordinate can represent a distinct category. For many other data sets, like the network traffic that we want to analyze, there are more than three categories.

There are two methods of dealing with this. First, the third coordinate can simply represent all other data, giving us two distinct categories, and a third catch-all category. Second, we can have three distinct categories, and anything that does not fit one of the three categories is removed from consideration. The Triangle visualization allows one to specify which values are to be used for each category.

A second significant limitation of trilinear coordinates is that each of the three categories must be independent of each other. Anything that fits in the $x$ category, must not fit into the $y$ or $z$ category. This restricts what kinds of categories can be chosen, to some degree, though most of the common categorizations should work without any trouble. For example, one could use the transport layer protocols TCP, UDP, and ICMP as the three categorizations. Anything that uses TCP cannot also use UDP or ICMP. However, one could not mixing transport layer and network layer protocols, and choose TCP, UDP, and IP protocols as the categorizations, because all TCP and UDP packets are also IP packets.

*Updating Nodes*

One particular challenge with this visualization is how a node’s trilinear coordinates gets updated, based on a new piece of data. A node contains a collection of data that determine the node’s trilinear coordinates. Because data within AdviseAid is
being processed as time passes, or at least in some form of sequential ordering, the trilinear coordinate of the node is the average of each of the individual data items over time. For example, if two computers are communicating with each other and they send four TCP packets and then a UDP packet, the node’s current location would be \((0.8, 0.2, 0.0)\). Thus with a basic implementation, a node’s trilinear coordinates is the average of the node’s data over its entire history.

This type of basic implementation, however, has two big drawbacks. First, because it requires all of the node’s history, as more time passes, more memory will be required of the visualization. Second, if a great deal of normal traffic occurs, and then suddenly a smaller amount of anomalous traffic appears, because there is so much more normal traffic, the node will barely change coordinates, and it will not produce a visible anomaly to the user.

An improvement on the basic naïve implementation is to only keep track of a user-specified number of recent packets, and average those instead, or alternatively, only keep track of all packets within a certain amount of time before removing them from the list. The former approach creates a maximum cap on the total amount of data stored, though the latter approach is likely to produce more accurate results, as it is time-based. It does not, however, guarantee a limited number of packets for each node, as the data set may contain a very large amount of data during a given timeframe. This data would be stored in its entirety, until enough time has elapsed, possibly causing the program to use up its available memory and slow the execution of the visualization.
While either of the two above approaches would be vast improvements over a naïve implementation that stores everything, neither of them were implemented. Instead a hybrid approach was taken.

The approach that was implemented in the current version of the Triangle visualization is a method wherein only a given number of packets are stored for the node, but the trilinear coordinates of the node is calculated based on both the value of the new packets, and older packets, combined together with a weighting or decay factor.

\[
t_{i+1} = \alpha t_i + (1 - \alpha) \sum_{k=1}^{n} v_k
\]

Where:
- \( t_i \) is the trilinear coordinate at the \( i \)th timestep,
- \( \alpha \) is the decay factor, and \( 0 \leq \alpha \leq 1 \),
- \( v_k \) is the trilinear coordinate of the \( k \)th new piece of data, and \( v_k \) is either \((1, 0, 0)\), \((0, 1, 0)\), or \((0, 0, 1)\),
- and \( n \) is the total number of new packets.

We found that a decay factor of \( \alpha = 0.995 \) works well, though this value is customizable, and other values may be better for different applications. In general, the closer \( \alpha \) is to 1, the slower the node will move to reflect recent changes. With \( \alpha = 1 \), the node would never move, and instead, retain its original position forever. The lower \( \alpha \) is, the faster it will move. If \( \alpha = 0 \), all old data is discarded immediately, and only the current data is used to determine the current location. This, in effect, degenerates to one of the previous methods described, wherein only a certain number of recent packets are used. Values near, but not at 1 tend to produce the best results.

This type of incremental change allows the visualization to be able to detect some forms of low and slow attacks. While old data may be completely thrown away, a
summary of the data tracked numerically until a great deal of new data arrives. For instance, if a port scan is attempting to go undetected by only scanning one port every day, the Triangle visualization may still be able to detect it, because even though the data from days earlier has been removed, it is still having an impact on the node’s location because of the decay factor.

*Screen Coordinates and Trilinear Coordinates*

In order to be plotted on the screen, a trilinear coordinate must first be converted from trilinear coordinates to Cartesian coordinates in the screen’s coordinate space. In Java’s Swing, as well as virtually all other windowing systems, the top left corner of the screen or window has the coordinates (0, 0). The x-axis goes left to right, and the y-axis goes from top to bottom.

In order to plot the point, we must be able to take trilinear coordinates and the given Cartesian coordinates of three points on an equilateral triangle to determine the screen coordinates of the given trilinear point. Additionally, the Triangle visualization allows the user to select nodes in the graph. Since mouse events in Java are referenced by the screen coordinates of the mouse, we also need to be able to convert from Cartesian coordinates to trilinear coordinates.

Mertie describes the process of converting trilinear coordinates to Cartesian coordinates [19]. The Triangle visualization follows the basic formulas Mertie provides, with a few variations and modifications to match the screen coordinate system that we are using.
In this section, we refer to the main triangle that contains all points in the plot as $T$, and each of the vertices of the triangle as $T1$, $T2$, and $T3$, where $T1$ is the top vertex, $T2$ is the right vertex, and $T3$ is the left vertex of the equilateral triangle.

To convert a 2D screen coordinate to trilinear coordinates, we follow a two-step process. First, the mouse coordinates $(m_x, m_y)$ are converted to a unitized box surrounding an equilateral triangle with sides of length 1. Next, we determine a number of important measurements on the original triangle.

The following formulas are used to determine the values of $L$, $h$, and $a$:

$$L = T3_x - T2_x$$

$$a = \frac{\sqrt{3}}{6} L$$

$$h = 2a$$

To calculate the point $(x, y)$, given $(m_x, m_y)$, we use the following equation:

$$x = T3_x - \frac{L}{2} + m_x$$

$$y = m_y + T3_y - a$$

We can then calculate the trilinear coordinates for the point $(\alpha, \beta, \gamma)$ with the following equations:
\[
\alpha = \frac{1}{3} \left[ \frac{2 (p_3 - x \cos \omega_3 - y \sin \omega_3)}{h} + 1 \right]
\]
\[
\beta = \frac{1}{3} \left[ \frac{2 (p_2 - x \cos \omega_2 - y \sin \omega_2)}{h} + 1 \right]
\]
\[
\gamma = \frac{1}{3} \left[ \frac{2 (p_1 - x \cos \omega_1 - y \sin \omega_1)}{h} + 1 \right]
\]

where \( p_1 = p_2 = p_3 = \frac{1}{3} \)

and \( \omega_1 = \frac{\pi}{3}, \quad \omega_2 = \frac{5\pi}{6}, \quad \omega_3 = \frac{3\pi}{2} \)

To convert from trilinear coordinates to Cartesian screen coordinates, we use the formulas below:

\[
x = + (T2_x - T3_x) \left( \frac{\beta}{1 - \alpha} \right)
\]
\[
y = T1_y + (T2_y - T1_y)(1 - \alpha)
\]

Note that \( \gamma \) is not used in these calculations, because \( \gamma \) is dependent on \( \alpha \) and \( \beta \).

**Zones and Trilinear Coordinates**

Zones in the Triangle visualization are small triangular regions within the main triangular plot area. All points within a zone have very similar trilinear coordinates, and as such, zones have an important role in helping the user determine when the behavior of a node becomes anomalous. A node crossing into a zone in which it has not spent much or any time is an indication that something odd may be going on. It is especially so if the
node makes a rapid change to a distant zone. A screenshot from the completed Triangle visualization, is shown in Figure 20.

The current implementation of the Triangle visualization stores the zones in an array of arrays. Each row in the array represents a horizontal strip of zones across the triangle. Each row contains alternating orientations of triangles, upward oriented, then downward oriented. The first row is of size 1, containing only the single upward-oriented zone at the top of the triangle, the second is of size 3, containing the three zones in the

![Figure 20. Screenshot of zones in the triangle visualization. In this image, the zone data is for the selected point in the lower right part of the triangle. The zones indicate where the node has spent most of its time, with the color of the zone changing from black, representing zones that the node has never been in, to blue, where it has spent a long period of time. If a node moves into a black area, it is clear that something anomalous is happening, and is worth further investigation.](image-url)
second row of zones (two upward- and one downward-oriented triangles), the third contains 5 total zones, and so on, until the bottom of the triangle is reached.

As a part of this process, it is fairly common to need to determine the zone of a node, given a trilinear coordinate. This problem can be summarized as this: given a trilinear coordinate \((\alpha, \beta, \gamma)\), and the total number of rows of zones in existence, determine the row and column in the triangular array for the zone that it belongs to.

To accomplish this, we first calculate the zone’s value along each of the three axes, using the equations below:

\[
\begin{align*}
x &= (h - 1) - (1 - \alpha)h \\
y &= (h - 1) - (1 - \beta)h \\
z &= (h - 1) - (1 - \gamma)h
\end{align*}
\]

*where \(h\) is the total number of rows in the zone array*

Then, to lookup the actual row and column from these values, we use these equations:

\[
\begin{align*}
row &= (h - 1) - x \\
column &= \begin{cases} 
2y & \text{if } y + z = row \\
2y + 1 & \text{if } y + z \neq row
\end{cases}
\end{align*}
\]

Note that the function for determining the column becomes a piecewise function to account for both upward oriented and downward oriented zones.
Results

We have run the Triangle visualization with an assortment of TCP dump files. The results are very interesting; they show that the visualization has the ability to detect various network security events and threats. In this section, we review results from three of the files, we discuss the results of the other goal of creating the visualization, and finally we discuss how the development process was assisted by the AdviseAid framework.

One interesting example that we noticed with the Triangle visualization is the detection of port scans. Port scans were easy to discover in the capture_2 file that ships with AdviseAid as an example data set, and is commonly used with other AdviseAid visualizations. The Triangle visualization shows port scans as nodes that continue to move from one vertex to the other and back. Occasionally, if the visualization is configured right, the node will go around each of the vertices in a cycle. Figure 21 shows a screenshot of the Triangle visualization with two port scans visible. On the right is a port scan that is easy to see because the node continues to move back and forth from the top corner to the lower right corner. Upon further drill down, one sees that the scan starts at around port 1000 and incrementing by one on each step, up until around 4000, when it starts over. In the lower left corner is a second port scan that started somewhere in the 2001-6000 range, and continued to climb, and eventually moved the node over to the left vertex.
Figure 21. Two port scans in the Triangle visualization. Each vertex of the triangle represents different ranges of ports, with the low end ports at the top vertex, the middle range ports on the right vertex, and the high range ports on the left vertex. The node marked with the red arrow continues to transition between the top and the right vertices during the duration of the file, as indicated by the light blue arrow. The node in the left corner has also just moved from the mid range ports to the high range ports. Both of these are clearly port scans in the file.

Near the end, several nodes suddenly turn extremely unstable, and move all over the plot. In particular, one node, shown in Figure 22, arcs through the middle of the plot indicating that it was communicating on a large number of ports. This shows a serious problem with the node. Because this node is selected, we can also see that this node had previously spent almost all of its time near the top vertex by looking at the marked zones in blue.
Figure 22. Anomalous activity shown in the Triangle visualization. This shows the same capture file, near the end as several nodes begin moving all across the plot in a significant anomaly. The purplish arc in the bottom left came from a node, now located in the lower left corner which represents the communication between two computers on a wide variety of ports, causing it to arc abnormally.

In a second example, shown in Figure 24, the visualization has the vertices mapped to packet size. In this example, the top vertex represents communication with small packets (less than 100 bytes long) the right vertex with medium sized packets (100 to 1000 bytes) and the left vertex represents communication with large packets (longer than 1000 bytes long) as would be normal during a large file transfer. During the analysis, we see two nodes that are constantly moving around near the lower middle part
Figure 23. The Triangle visualization mapped to packet size. Two nodes near the bottom center are moving around together, indicating that they are communicating with each other. Another node suddenly moves from the top vertex out to the middle, because it has suddenly started sending much larger packets. This is likely a sign that the connection the node represents has started transferring large amounts of data.

of the triangle. They move around together, which indicates that the two nodes are likely communicating with each other.

Another interesting discovery in this example is the node in the center that has moved from the top vertex down to the lower middle of the plot. The node is selected, and so by the zones it is clear to see that the node had spent a great deal of time sending only small packets, and suddenly, it began sending much larger packets. The connection it represents has clearly changed the type of data it is sending from small single packets
to larger packets. This is likely the beginning of a file transfer that will produce bigger full Ethernet packets.

A third and final example shown in Figure 25 shows a network capture from a single computer. The single computer was accessing the Internet to watch a streaming video. After running only the streaming video, another single web page was opened, producing a collection of other nodes. Upon detailed investigation, it is clear that the collection of nodes are coming from the original page, as well as other sites that are providing additional content, like images, ads, and so on. The two purplish nodes heading towards the lower left corners have likely begun transmitting larger files, like images or Flash objects.

Figure 24. The Triangle visualization with video and web traffic. This image shows a streaming video connection in the bottom left, and a large collection of connections transferring small packets. The large collection of nodes appeared when a single web page was opened. The multiple nodes are showing all of the content from other pages that were needed to load the complete page, like ads, images, and so forth. The two purplish nodes heading towards the lower left are likely transmitting images or Flash objects.
Components of AdviseAid Utilized

One of the key goals in the development of the Triangle visualization was to illustrate that AdviseAid provides a large collection of tools speed the development process along. It is therefore worth discussing briefly the components of AdviseAid that have proven useful during the development of the Triangle visualization, to illustrate how AdviseAid can assist in the creation of visualization prototypes.

The Triangle visualization relied heavily on the default input parsers that AdviseAid provides. In particular, the Triangle visualization took advantage of the Jpcap parser, which reads TCP dump files. Since the goal with the Triangle visualization is to prototype a new novel visualization technique, it saved a great deal of time to be able to skip the file reading and parsing entirely, and just work with the data pipeline that AdviseAid provides to the visualizations.

The Triangle visualization also took advantage of the visualization framework that was assembled years ago, and greatly simplified in the more recent months. Essentially, the entire development of the Triangle visualization was devoted initially to the design, the implementation mostly devoted to the visualization’s update() and draw() methods, where the underlying data model was updated, expanded, and otherwise modified to reflect the newest collection of data supplied by AdviseAid, and to the actual rendering of the visualization. Because AdviseAid took care of the event loop, our visualization did not need to worry about that aspect either.

Along the same lines, the framework that AdviseAid provided for doing rendering with Swing allowed us to use a programming technology that is widespread and well
known, but extremely easy to use for our rendering. Because Swing rendering is so strongly supported with AdviseAid, we did not have to waste a great deal of time working with the more tedious (though more powerful) OpenGL capabilities that AdviseAid also supports.

We were also able to take advantage of a number of GUI components and structures that appear in the main AdviseAid window to interact with the user. AdviseAid’s mapping system let us easily get feedback from the user to try different mappings. The user was able to decide what each of the points on the triangle represented, and change it to get a different view of the data set. While not all mappings are useful in the Triangle visualization, the built-in functionality of AdviseAid easily allows one to experiment with all possible options. This allowed us to quickly and easily learn important lessons from the prototype that would be needed in a final implementation of the visualization.

Lastly, the screenshot and video capturing tools of AdviseAid made it extremely easy to put together this thesis, as well as a collection of other reports about the visualization. While screenshots and video captures can be generated in other ways, the native ability of AdviseAid to do this meant that we did not need to spend time working with either an image editor or a video editor, and we did not need to spend any extra money in the development to get quality screen and video capturing.

Overall, we saw that building on the AdviseAid framework provided us with a great deal of boilerplate code and functionality that allowed us to skip the low-level basics of producing a visualization, and move ahead with the implementation of the
visualization itself, giving us extra time to add in additional improvements, such as the zones.

**Future Work and Conclusions**

The results show that the Triangle visualization is capable of detecting a variety of network attacks and significant events. With continued development, the visualization could prove to be even more powerful. We have seen that using anomaly-based visualization can alert the user to when something is amiss on their network. We have shown that port scans are easily visible in the visualization, as are sudden changes in the behavior of a computer. These events allow a network analyst to quickly see what may be wrong on the network.

One of the best features of the Triangle visualization is that it is has proven to be easy to use. Many other visualizations are done in three dimensions, which requires the user to rotate the display around to understand what is going on. A simple 2D display like the one used in the Triangle visualization allows a user to be able to analyze the network without having to work with complicated user interactions to control an abstract 3D environment. A user can tell what might possibly be wrong by simply looking at things that are changing in the display, especially the movement of nodes. “Normal” nodes stay put, while others move around.

The Triangle visualization has one significant limitation that will most likely not be able to be fixed. The fact that all pieces of data must be categorized into one of exactly three categories is the driving force behind the underlying ternary plot, but it is also an intrinsic limitation to the plot and to the visualization as a whole.
There is another limitation that the Triangle visualization has that is common to all anomaly-based systems: a system can perform bad activity so often that malicious events become normal. This is not likely to occur in a real world situation, however, and for now, we are ignoring this concern.

Scalability also appears to still be a significant issue with this visualization technique. Like many other visualizations, a large data set greatly clutters the screen, which restricts what the user can do with it. Different visualizations handle this to varying degrees of success, but the Triangle visualization is probably somewhat worse than average in this regard. There may be things that can be done to improve this problem, however. First, since the entire visualization is based on anomalous behavior, a metric of anomalous behavior in a node could be provided. It would also be beneficial to add a transparency component to each node, so that nodes that are behaving within normal bounds are completely transparent, and nodes that are an anomaly appear opaque to varying degrees, depending on how anomalous it is. This would ensure that only anomalous data is displayed, and would greatly reduce clutter on the screen. A feature such as this is going to be almost mandatory for this visualization before it could be considered viable for commercial uses.

There is a small collection of other areas in which this visualization could be improved if additional time could be put into it. More mappings would help expand the power of the visualization and allow it to be applied to a wider variety of input sources.
There are also many parameters the visualization tracks that the user cannot modify, such as the node size, or the zone color. A developer can change these values in code, but it would be beneficial to end users to have access to these features.

Another possible improvement to the interface would be to enable the size of nodes to increase and decrease, based on the total amount of data being processed at the given moment. The Triangle visualization currently analyzes the type of data each node is processing, and almost completely ignores the volume of data. Adding in this feature would greatly improve its power. If this feature were implemented, though, it would be important for it to show the change in volume of data, rather than the actual amount of data, in order to allow the visualization to focus on anomalous behavior. For example, a popular web server like google.com handles a vast number of requests every second, while a single desktop computer only handles a very small amount of traffic. It would be important for this feature to show that Google’s server has suddenly begun transmitting an abnormally small amount of data, or that an individual’s desktop computer is suddenly being flooded with data, instead of the normal slow trickle. Thus, the size of the node must be based on how abnormal the volume of data is, rather than a measurement of the total volume.

This case study shows that AdviseAid provides many powerful features that allow complicated visualizations to be prototyped in a short period of time by providing the user with necessary framework to work quickly and effectively. It also illustrates that the Triangle visualization is a relatively effective technique by itself. There are still many ways that it can be improved, and it has a few limitations that will not be able to be
addressed, simply because of the basic requirements of the underlying ternary plot, but overall the visualization is powerful and has been shown to be able to detect a variety of network based attacks.
REFERENCES


