"Computing Platform Coverage via Light Host-based Intrusion Detection"

June 2002

Final Technical Report

CDRL A004

Cigital
{P.O.C. Matthew Schmid, mschmid@cigital.com}

20020625 080

Computing Platform Coverage via Light Host-based Intrusion Detection

Contract No. DAAH01-00-C-R222
Final Technical Report
Cigital[*]

1 Executive Summary

Malicious software and hostile intrusions represent two of the largest threats to information systems today. During the course of this project we developed solutions that address both of these issues. The BayeScan prototype is a novel approach to detecting malicious software before it has a chance to strike. The AppID prototype provides a framework for the development of real-time host-based intrusion detection technology. The creation of these tools provides users with access to leading-edge research technologies in the fight against malicious software and malicious attackers.

Today's commercial antivirus software is based largely on pattern recognition techniques. While effective at identifying known malicious software, this approach is notoriously poor at finding novel attacks. This leaves computer systems highly vulnerable to newly created viruses and Trojan horse programs. To address this problem we explored a number of standard data mining techniques to develop an accurate classifier for previously unseen programs. The classification techniques that we explored were rated by their ability to accurately identify an unknown executable as being malicious or benign. The most successful classification technique that we explored was Naïve Bayes classification. Our successful experiments with this approach led to the development of the BayeScan and Malicious Email Filter prototypes. BayeScan implements a graphical user interface on the Windows platform to enable users to easily scan executables for sign of maliciousness. The Malicious Email Filter integrates with the UNIX procmail system to provide gateway protection against malicious software.

The steady increase in penetration attacks against computers running the Windows operating system, combined with the heavy reliance of government and commercial clients on these machines, led us to revisit some of the research conducted during phase one of this contract. During phase one we had experimented with using Cigital's intrusion detection technology to detect attacks against Windows applications. On this contract we developed a framework for extending the phase one work to perform real-time host-based intrusion detection for Windows NT/2000 systems. This framework, known as AppID, enables researchers to quickly develop and evaluate intrusion detection algorithms for Windows platforms.

We are delivering to the government the BayeScan, Malicious Email Filter, and AppID prototypes. We have included full source code to these tools as well as additional documentation (see the *BayeScan User Guide* and *AppID User Guide*). The BayeScan and AppID prototypes are also delivered in binary form and packaged for easy installation.

---

[*] The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

2 Introduction

Two of the greatest threats to information systems today are damage caused by malicious software and attacks by malicious individuals. During the course of this project we developed solutions that address both of these issues. The BayeScan prototype is a novel approach to detecting malicious software before it has a chance to strike. The AppID prototype provides a framework for the continuing development of intrusion detection technology. While neither of these tools provides a complete solution to the problem, they each represent advances in the fight against malicious software and malicious attackers.

2.1 The malicious software threat

New malicious software is developed every day. Virus writers modify existing viruses to be more dangerous and less detectable, or develop novel attacks that make the job of the antivirus expert increasingly difficult. In recent years the expanding presence of the Internet and the predominance of the Windows operating system contribute to a fertile environment for the spread of malicious software. Computer viruses benefit from the homogeneity of the operating systems connecting computers around the world. Just as most commercial software will run equally well on all Windows platforms, so will most malicious programs.

Most antivirus software is reactionary by nature. The most common method of detecting malicious software is through software scanning. Antivirus software analyzes a program in an attempt to determine the presence or absence of known instruction patterns that have been determined by antivirus experts. Over the last decade significant improvements have been made to antivirus software. The scanners are faster, more robust, and better able to determine inexact pattern matches. They have also been improved to handle encrypted or polymorphic viruses. The best-regarded virus scanners are capable of detecting all of the viruses on the so-called *WildList* [1] in addition to thousands of less common threats.

One major problem remains to be overcome. Commercial antivirus software is notoriously bad at detecting *new* malicious software. New malware contains patterns of instructions that have never been seen before and may not contain any of the signatures of known viruses. The speed with which many viruses spread means that many hosts can be attacked before antivirus companies release a solution. This window of attack leaves computers vulnerable to a barrage of new attacks.

On this project we have developed a technology that can help to detect new, unknown, malicious software. The prototype that we developed, known as BayeScan, uses data mining techniques to extract characteristics from an executable and to classify that program as malicious or benign. Experiments conducted by Columbia University illustrate the viability of this approach [2].

2.2 The intrusion threat

In addition to the malicious software threat, the number of remote attacks against Windows machines has also risen significantly in the last few years. These attacks can be initiated by an individual, or, as in the case of the Code Red attacks, they can propagate among computers without human interaction. In either case the approach is the same: identify a process on the remote machine that is accepting connections and exploit it to perform actions on that machine.

Cigital has been developing technologies for detecting intrusions for the last several years. The approach that we have taken is to perform host-based anomaly detection. The basic idea of anomaly detection to learn an application's normal behavior, then identify deviations from this normal behavior. This approach is not unique to Cigital – it has been studied at various institutions around the globe [3,4,5,6]. Our intrusion detection techniques, however, have demonstrated strong performance in evaluations including the Lincoln Laboratories intrusion detection evaluation in 1998 [12].

A major obstacle to deployment of our system was the need for an intrusion detection architecture that could collect and process system data in real-time. Another issue was the increasing need for intrusion detection systems that would run on Windows operating systems. We utilized resources on this project to develop an architecture that supported the real-time processing of Windows system data. This provides a framework for the continued development of intrusion detection tools that can be rapidly deployed in real environments.

3 Detecting malicious software

The approach that we are using to detect malicious software was built on the research done by our partner in this contract, Columbia University. The six-month technical report (CDRL A003) discusses this work in detail. The latter half of this contract was primarily a development effort to transition the research idea into a tool that would be useful to both antivirus researchers and end users. To meet this goal we developed a testing framework that facilitates malicious software detection experimentation and a graphical user interface designed for the end-user.

3.1 Research overview

Our approach to this problem was to explore a number of standard data mining techniques to develop an accurate classifier for previously unseen binaries. We gathered a large set of programs from public sources and separated the programs into two classes: malicious and benign executables. The classification techniques that we explored were rated by their ability to accurately identify the appropriate category for an unknown executable.

The framework supports different methods for feature extraction and different data mining classifiers. We used system resource information, strings, and byte sequences that were extracted from the malicious executables in the data set as different types of features. We experimented with both an inductive rule-based learner that generates boolean rules based on feature attributes, and a probabilistic method that generates

probabilities that an example was in a class, given a set of features. This report focuses on our use of a Naïve Bayes classification system because it produced the most promising results. For a full description of the experimental process see [2].

3.1.1 Naive Bayes

The Naive Bayes classifier computes the likelihood that a program is malicious given the features that are contained in the program. We will describe our experiments using the Naïve Bayes algorithm with the features extracted using the *strings* program to perform a classification.

In [13] the authors performed a similar experiment when they classified text documents according to which newsgroup they originated from. In this method we treated each executable's features as a text document and classified based on that. The main assumption in this approach is that the binaries contain similar features such as signatures, machine instructions, etc.

Specifically, we wanted to compute the class of a program given that the program contains a set of features. We define C to be a random variable over the set of classes: benign, and malicious executables. That is, we want to compute $P(C \mid F)$, the probability that a program is in a certain class given the program contains the set of features F. We apply Bayes rule and express the probability as:

$$P(C \mid F) = \frac{P(F \mid C) * P(C)}{P(F)}$$

Equation 1

To use the naive Bayes rule we assume that the features occur independently from one another. If the features of a program F include the features $F_1, F_2, F_3, \ldots F_n$, then Equation 1 becomes:

$$P(C \mid F) = \frac{\prod_{i=1}^{n} P(F_i \mid C) * P(C)}{\prod_{j=1}^{n} P(F_j)}$$

Equation 2

Each $P(F_i \mid C)$ is the frequency that string F_i occurs in a program of class C. $P(C)$ is the proportion of the class C in the entire set of programs.

The output of the classifier is the highest probability class for a given set of strings. Since the denominator of Equation 1 is the same for all classes we take the maximum class over all classes of the probability of each class computed in Equation 2 to get:

$$MostLikelyClass = \max_C (P(C) \prod_{i=1}^{n} P(F_i \mid C))$$

Equation 3

In Equation 3, we use $\max_{c}$ to denote the function that returns the class with the highest probability. *Most Likely Class* is the class in C with the highest probability and hence the most likely classification of the example with features F.

To train the classifier, we recorded how many programs in each class contained each unique feature. We used this information to classify a new program into an appropriate class. We first used feature extraction to determine the features contained in the program. Then we applied Equation 3 to compute the most likely class for the program.

3.1.2 Performance

We estimate our results over new data by using 5-fold cross validation. Cross validation is the standard method to estimate likely predictions over unseen data in data mining techniques. For each set of binary profiles we partitioned the data into 5 equal size groups. We used 4 of the partitions for training and then evaluated the rule set over the remaining partition. Then we repeated the process 5 times leaving out a different partition for testing each time. This gives a reliable measure of our method's accuracy over unseen data. We averaged the results of these five tests to obtain a measure of how the algorithm performs over the entire set.

To evaluate our system we are interested in several quantities:

- True Positives (TP): the number of malicious executable examples classified as malicious executables
- True Negatives (TN): the number of benign programs classified as benign.
- False Positives (FP): the number of benign programs classified as malicious executables
- False Negatives (FN): the number of malicious executables classified as benign binaries.

We were interested in the detection rate of the classifier. In our case this was the percentage of the total malicious programs labeled malicious. We were also interested in the false positive rate. This was the percentage of benign programs that were labeled as malicious, also called false alarms.

The Detection Rate is defined as $\dfrac{TP}{TP + FN}$, False Positive Rate as $\dfrac{FP}{TN + FP}$, and Overall Accuracy as $\dfrac{TP + TN}{TP + TN + FP + FN}$.

3.1.2.1 Results from Columbia work

The initial data set consisted of a total of 4,266 programs split into 3,265 malicious binaries and 1,001 clean programs. There were no duplicate programs in the data set and every example in the set is labeled either malicious or benign by the commercial virus scanner.

The malicious executables were downloaded from various FTP sites and were labeled by a commercial virus scanner with the correct class label (malicious or benign) for our method. Five executables in the data set were Trojans and the other 95 consisted of viruses. Most of the clean programs were gathered from a freshly installed Windows 98 machine running MSOffice 97 while others are small executables downloaded from the Internet. The entire data set is available on the Columbia University website (http://www.cs.columbia.edu/ids/mef/software/).

The results of these experiments are presented in Table 1.

Classifier	True Positives	True Negatives	False Positives	False Negatives	Detection Rate	False Positive Rate	Accuracy
Naïve Bayes (initial data set)	3176	960	41	89	97.43%	3.80%	97.11%

Table 1: Initial data set results

The Naive Bayes algorithm using strings as features performed the best out of the learning algorithms and better than the signature method in terms of false positive rate and overall accuracy (see Table 1). It is the most accurate algorithm with a 97.11% detection rate.

3.1.2.2 Revised data collection

During a second set of experiments we restricted our data set to a collection of currently active malicious threats. The data that we are using for training and testing in this experiment consists entirely of 32-bit Microsoft Windows binaries. The benign programs in our test set are a combination of those taken from a Windows NT 4.0 installation and those found on the Internet. There are 725 benign files in our data set. The malicious programs that were used during this phase of the experiment consist of first generation 32-bit viruses collected from the Internet. There are 236 malicious files in our data set.

Classifier	True Positives	True Negatives	False Positives	False Negatives	Detection Rate	False Positive Rate	Accuracy
Naïve Bayes (W32 data set)	188	695	30	48	79.67%	4.14%	91.88%

Table 2: Win32 results

Though smaller, this data set is more representative of current threats against Windows systems. The reduction in the detection rate and accuracy is likely due to having fewer examples in our training set. As the number of malicious threats continues to grow we will expand our training set to include these new examples.

3.1.3 Improvements

We experimented with a variety of minor changes to the techniques used for feature extraction with the goal of improving the effectiveness of our tool. These changes resulted in slight improvements to the detection rate and overall accuracy.

3.1.3.1 Unicode

The GNU strings program does not handle Unicode strings. Unicode strings are formed using two bytes to represent each character instead of just one as with ASCII strings. Some Windows applications make use of Unicode strings. In this experiment we have tweaked the feature extractor to extract Unicode strings and to include these in the detection model. It is important to note that this feature extractor also extracts strings from the entire file. The other settings are the same as those in the baseline experiment.

3.1.3.2 Lengths

In the baseline experiment we extracted any strings that consisted of four or more printable ASCII characters. In this experiment we explored the effects of varying the minimum string length.

3.1.3.3 Unique strings

The baseline system made use of each feature that it was given, regardless of whether or not it had seen that feature before. We wrote a feature extractor that would extract only one instance of each string. If a string appeared more than one time in a file the classifier only counted it once.

3.1.3.4 Entire file

By default, the GNU *strings* program only extracts strings that are located in the code section of a program. We explored the effects of extracting strings from all parts of the binary.

3.1.3.5 Results

Table 3 shows the results of combining all of our successful tweaks into one classifier. Scanning the entire file for strings is an improvement over just looking at the code section of the program. Extending the feature extractor to include Unicode strings had a marginal impact on performance. Looking at only unique strings resulted in worse performance than the original classifier. Combining several of these improvements did produce a classifier that is slightly more accurate than the original.

Classifier	True Positives	True Negatives	False Positives	False Negatives	Detection Rate	False Positive Rate	Accuracy
Naïve Bayes (W32 data set)	192	708	17	44	81.36%	2.34%	93.65%

Table 3: Improved Win32 results

3.1.4 Limitations

Although these experiments demonstrate promising results they are not necessarily indicative of real world performance. Here we discuss additional experimentation that needs to be performed.

The malicious programs used in training were raw viruses not infected programs. In most cases, an executable virus is a program in itself. This program contains the logic

necessary for the virus to locate a suitable host program and infect that host in such a way that when the host is executed it will invoke the virus code. When viruses are encountered in the wild they are usually already attached to a host program. This is the virus' method of disguising itself and therefore continuing its existence.

From the perspective of this experiment, the primary difference between raw viruses and infected executables is that infected executables will contain a large number of benign characteristics as well as the malicious characteristics from the virus. As the system is currently trained, it is likely to identify an infected executable as benign due to the large number of benign characteristics that it will find.

This problem needs to be addressed through the introduction of more training data. The training data should consist of benign executables and benign executables that have been infected by viruses. We were unable to obtain the quantity of data necessary to conduct a statistically valid experiment that met these criteria. Infecting executables is time-consuming and sometimes fruitless exercise. Viruses have an infinite number of infection methods at their disposal, and it is not always clear what will trigger them. The statistical classifiers that we are using require large amounts of data to function accurately, and unfortunately we do not have this data at our disposal.

We believe that the classifiers that we used during this experiment can be adapted to work on the suggested training data. The result of training on infected executables and benign executables should be a classifier that learns that benign characteristics are not statistically important while malicious characteristics (or the absence of malicious characteristics) are the key factors in the classification.

Another aspect of training on infected executables is that some viruses decrypt the bulk of their functionality on the fly. An encrypted virus that changes its encryption key will not have string characteristics that we can identify via static analysis. Although many encrypted viruses do exists, many more lack this stealth. The classifier used in our experiments will be effective against unencrypted viruses.

3.2 Development

The data mining approach to detecting malicious software outlined above led to the development of three related tools. The first is a framework for conducting experiments and tuning the algorithms and selection of features. The second is the BayeScan prototype. This is a Windows based tool with a graphical user interface that enables end-users to easily analyze suspicious executables. The final tool is the Malicious Email Filter (MEF), which is an incorporation of our classification technology into a gateway email scanner. This section describes the key aspects of these tools.

3.2.1   Testing framework

The testing framework that we developed enables researchers to easily make changes to the existing malicious software detection system and to analyze the performance of these modifications. It simplifies the management of a malicious software collection and the running of experiments.

3.2.1.1 Architecture

This section describes the role of the classes that are essential to the running of an experiment.

Configurations

The configuration class manages the data contained in a configuration file for use by the other classes. In addition to specifying the location of data files for the classification model and testing, the configuration file also allows the user to toggle options such as uniqueness or filtering. Appendix A contains a summary of the options controlled using configuration files.

Models

The role of the model is to extract data from executables and to manage this data. The base model is a pure virtual class whose implementation is the responsibility of the subclasses. An example of a frequently used model in our experimentation is the *OnlineStringModel*. This model extracts strings from an executable and maintains a database containing the frequency of these strings in benign and malicious programs.

Classifiers

All classifiers inherit from the virtual class *ClassifierBase*. The classifier implements the algorithm that distinguishes between malicious and benign executables. It uses data contained in a model class to classify a file that is supplied as an argument.

Filters

A classifier may choose to use a *filter* class to filter the data that is passed to the classifier's algorithm. An example of a filter class is the *BenignOccursOnce* class that is used to cause a classifier to ignore any strings that occur with a frequency of one in our data set of benign programs.

3.2.1.2 Running an analysis

The analysis program produces a series of output files containing the classification results of the various classifiers implemented within the application. Each output shares a common format. The first line of each results file contains two numbers. The first number is the number of training benign files, and the second number is the number of training malicious files used to construct the classification model. For example, the first line of scale5.0 is "551 178". The remainder of the results file contains the scores for each file in the test set. After the first row, each row of the results file lists the following information: 'v' or 'b' to indicate the actual class of the test file (virus or benign); the benign score; the malicious (virus) score; and the name of the test file.

For some classification algorithms only the benign score is given a value – the malicious score is listed as '0'; For these result files, the benign or virus classification is dependent on a range being developed for both classifications. As an example, the benign file may have a score less than 500, and all files with whose score is 500 or greater would be classified as a virus. The single score method is used for the following classification algorithms: *Bytes, Bytesfound, Items, Itemsfound,* and *Percentage.*

The naming convention for the output files is:

<algorithm_name>[min_feature_size].<cross-validation_set#>

For instance, the output file for the first cross-validation set using the classification *Minimum String Length Scaled* algorithm and a minimum string length of 10 would be "scale10.0".

3.2.1.3 Analyzing results using the accuracy program

The *accuracy* program is used to provide automated analysis of the accuracy of a specified classifier or classification algorithm. For the specified result files from the *analysis* program, the program creates a spreadsheet showing the overall accuracy of the classifier and the affect of a range of thresholds on the accuracy. The range of threshold values used is determined by the *accuracy* program based on the benign and malicious scores in the results files. For each threshold value in the range, the spreadsheet produced by the program lists a detection rate and a false-positive rate.

The *accuracy* program is a command-line executable. The format used for executing the program is as follows:

```
accuracy  <file>  <low #>  <high #>  [yes]
```

- *file* is the name of the results files to analyze minus the cross-validation set number.
- *low #* is the low number of the cross-validation set, usually '0'
- *high #* is the high number of the cross-validation set, usually '4'
- *yes* produces detailed output to be printed to the command line window.

For instance, for the *Minimum String Length Scaled* algorithm where '5' is the minimum string length, the analysis program produces results files scale5.0, scale5.1, scale5.2, scale5.3, and scale5.4. The command line for executing the accuracy program on this set of results files is "accuracy scale5. 0 4 ".

The *accuracy* program produces an .xls spreadsheet file and a .txt text file. The name of the output files corresponds to the results files used as input. For the example above, the *accuracy* program will produce "scale5.0-4.xls" and "scale5.0-4.txt". Sample output is shown below.

#False Positives	Detection	Threshold
4.05797	86.0119	-152.582
3.76812	86.0119	-145.515
3.04348	86.0119	-138.449
2.75362	85.1786	-131.382
2.31884	85.1786	-124.316

3.2.2 BayeScan prototype

The BayeScan prototype provides a graphical user interface that enables a user to analyze suspicious executables. The prototype implements the Naive Bayes classifier described above using the *Minimum String Length Scaled* algorithm with a minimum string length of 5. The classification model is comprised of strings extracted from the collection of malicious first-order 32-bit executables – viruses and trojans (see section 3.1.2.2) – and a collection of benign Win32 programs from a typical Windows 2000 workstation.

3.2.2.1 Installation

An InstallShield wizard is provided for installing BayeScan on a user's desktop. To install BayeScan, simply double-click the setup executable (setup.exe) in the BayeScan folder on the CD, or run setup.exe from the Start Menu's Run command. By default, all files associated with the application are installed in a folder named "BayeScan" inside "Program Files\Cigital" on the desktop's c: drive, but the user has the option to change the location during installation. In addition to the executable, the model file (BayeScan.mdl) and a corresponding hash file (BayeScan.hsh) are installed. The installation program also sets register settings for the location and name of the model file as well as the threshold used by the program. Refer to the BayeScan User's Guide for more information on the installation (Appendix B).

3.2.2.2 Executable analysis

BayeScan provides analysis of untrusted executables by comparing the features (strings) in the file to features in the BayeScan classification model. For each feature in the executable that is found in the model, a benign score and a malicious score is updated corresponding to the number of times the feature appears in the files that comprise the model. After examining all strings for an executable, the malicious score is subtracted from the benign score, and the threshold is added to the difference to determine the normalized score. A negative score indicates that the executable is believed to be malicious and a positive score indicates the executable is believed not malicious. The more negative the score (-100 is more negative than –20), the higher the likelihood the classification is correct. Likewise, the greater the score, the more likely a file is not malicious.

3.2.2.3 Updating models

The classification model can be updated in three ways: add a new executable to the model; add multiple executables to the model; merge a different model to the existing model.

When adding data from executable files to the model, BayeScan first compares the hash of each new executable to the hash of each file already contained in the model. The hash for each file used to make up the model is stored in the hash file with the same name as the model file. If the hash of the new file matches a hash in the hash file, BayeScan does not add the string data from the new file in the updated model since it already exists.

BayeScan allows the user to add feature data from a new executable to the model. The new executable can be benign (e.g., a new application recently installed on the

workstation) or malicious (e.g., a new virus). Additionally, BayeScan allows the user to add more than one executable at a time to the existing classification model. To add data from multiple files, a list file (.lst) with the full path name of the executables to add to the model must be created. The list must contain only one type of file -- malicious or benign.

The *merge models* functionality is designed to merge distinct models. For instance, if a model is built for a new class of files that were previously not in the original model, the merge functionality is an effective way to expand the range of protection provided by BayeScan. *Merge models* is also the method used for updating the model for a new group of viruses or benign executables that have been analyzed to form a new model -- effectively a "BayeScan update" model.

3.2.3   Malicious email filter

The malicious email filter (MEF) was developed by Columbia University. A full description of the work can be found in [7]. The MEF provides detection of known and unknown malicious email attachments using detection models obtained by data mining over known malicious attachments as filters. Combined with a central server, the malicious email filter provides for automated propagation of models used for detecting malicious attachments. Additionally, the filter included a system to monitor the spread of malicious attachments.

The malicious email filter is a network level application that resides on a UNIX server. As email enters the system, the tool filters Windows binaries that are attachments to email, and can wrap malicious email attachments. This method has the potential to stop malicious executables entering the network. This approach also removes the task of downloading virus scanner updates from the end-user, and places the responsibility on the system administrator for updating the filter's detection models.

Since the methods employed by the malicious email filter are probabilistic, the filter can detect a binary that is considered borderline. A borderline binary is a program exhibiting similar probabilities for both benign and malicious classes. If a binary is considered borderline, then there is an option in the network filter to send a copy of the malicious executable to a central repository such as CERT, allowing the binary to be examined by human experts.

3.2.3.1   Incorporation into Procmail

The malicious email filter examines email attachments by replacing the standard virus filter in Procmail with the detection model generated thru data mining. The UNIX-based email server extracts attachments using Procmail. The current supported email server is *sendmail.*

After an attachment is detected, the tool decodes the binary and examines it via data mining. The binary's byte strings are compared to the byte-sequences in the detection model and the probability of the string being malicious is calculated. If the probability of being malicious is greater than the probability of being benign, the binary is classified as malicious. Otherwise, the binary is classified as benign. If the attachment is classified as

benign, Procmail allows the email to pass to the recipient. If the attachment in instead classified as malicious, the recipient is warned that the attachment is malicious.

3.2.3.2 Borderline Classification

As stated earlier, it is possible for an attachment to exhibit near equal amounts of malicious and benign byte strings. In the event of an attachment being called borderline, a human expert must be used to make the determination of the binary's malicious or benign classification. Once the attachment is classified, its byte strings should be added to the detection model to aid the probability that a similar binary will be classified correctly using the malicious email filter. As the data set for the detection model grows, the likelihood of false positives and false negatives is reduced.

3.2.3.3 Updates to the Detection Model

In order to continually make the detection model better and more complete, it was necessary to create a method for adding new classified data to the model from formerly unclassified binaries. Updates to the model are accomplished easily due to the format of the detection model. The model is comprised of unique byte strings along with a binary and malicious score. The benign score is simply the number of times that the byte string appears in a benign binary, and the malicious score is the number of times that the byte string appears in a malicious binary.

Therefore, to add data from a newly classified binary (e.g., a borderline binary that has been classified by human experts) to the existing detection model, the byte strings from the newly classified binary are extracted and added to the model. If a byte string from the new binary is unique (i.e., not in the current detection model), it is added to the model and the correct score, binary or malicious, is given a value of one. The other score is given a value of zero. Otherwise, for each byte string in the new binary that is also in the detection model, the appropriate score is incremented by one.

Encrypted email is used to disseminate detection models. Once a new model is received, the Procmail filter can automatically update the detection model. Emails containing the new models can be addressed and formatted appropriately to allow the automated update.

4 The AppID intrusion detection framework

The AppID framework was designed to support the research and development of real-time intrusion detection tools. The design is flexible enough to be used by many host-based systems. During this effort we developed additional support for the collection and processing of Windows system data. The result of our work is a framework that facilitates future research and evaluation of intrusion detection algorithms.

4.1 Cigital intrusion detection

Cigital's intrusion detection technology is based on anomaly detection. Anomaly detection consists of two phases: (1) A training phase, where the normal behavior observed on an information system is used to automatically construct a model of normal behavior, and (2) a detection phase, where the behavior of the information system is

monitored in the field, and deviations from the original model of normal behavior are reported as signs of a possible intrusion.

Cigital's intrusion detection work, to date, has focused on host-based approaches. Certain aspects of each application's behavior are captured and used to characterize its behavior for the purposes of anomaly detection. Often, the captured data represents a series of calls to operating-system functions, though it should be noted that this is not a requirement. Cigital's anomaly detection technology simply requires a behavior stream, which consists of a series of symbols characterizing the behavior of something, and that thing may be an application, a resource, an interaction between two objects, and so on.

One way we can create an anomaly detector that judges whether behavior is normal or abnormal, within a certain context, is to build finite-state machines that model normal behavior. Recall that a finite state machine (FSM) consists of states and labeled transitions between states. When an FSM sees an input symbol, it looks for a transition out of its current state whose label corresponds to this symbol. If such a transition is found, the FSM follows that transition and enters a new state, but if no transition is found then the FSM is said to reject the entire sequence of symbols.

At any given time, the set of allowable transitions depends on the state that the FSM is currently in. The current state encodes the series of transitions that were taken to get to that state, so, in some sense, it encodes the long-term history of the application's behavior as well as the short-term history. Therefore, it is appealing to use FSMs as models of program behavior.

Cigital's integration of vocabulary extraction methods with the FSM has produced a more robust intrusion detection system. The principle of vocabulary extraction can be concisely expressed as follows: we automatically find common idioms in behavior traces so that those behavior traces can be represented more compactly. In our current approach, the idioms are repeated sequences of symbols in the behavior data, and sequences of symbols that always occur together. The immediate benefit of vocabulary extraction is that it can serve as a method of eliminating false positives generated by other anomaly detectors. This is because our approach leads to the creation of regular expression parsers that recognize repeated subsequences and indivisible substrings when they occur in new behavior data. If a substring has been accepted by one of these parsers we can regard it as normal — after all, the parsers are created on the basis of what was observed in normal behavior data — and if another intrusion detector sees an anomaly within such a substring we can ignore that alarm. With this improvement, we found that we could attain a reduction of about 30 percent in the false-alarm rates of the detectors in our experiments.

A method for automatic tuning of the IDS further improves the performance. Our approach uses the following steps:

1. The user specifies an acceptable false-alarm rate, which will be the same for all detectors.

2. For each application that is going to be monitored, the system builds a series of increasingly precise detectors. This continues until the user-specified false-alarm rate is exceeded, at which time the system outputs the last detector that did not exceed the threshold

3. If the detector for some application is too weak according to some heuristic estimate, then the user is warned that attacks on that application are unlikely to be detected. The user can then supply more training data (if desired), perhaps using automatic test data generation or by monitoring live applications.

The benefits of automatic tuning are quite dramatic on the Lincoln Labs data. We found that all attacks in the range of our detectors could be detected with a false-alarm rate of around *seven one-thousandths of one percent*. This is an improvement of over two orders of magnitude over our original results.

4.2 Development

The intrusion detection techniques described in section 4.1 were all evaluated through a process of off-line testing. System data was gathered ahead of time, pre-processed, and used during experimentation. Actual deployment and use of an intrusion detection system based on these techniques requires that data be processed as it is generated. The AppID intrusion detection framework was developed to provide this capability.

4.2.1 Collecting system data

As discussed above, host-based intrusion detection tools are often based on the analysis of local system data. Some of this data is available through built-in operating system mechanisms, but additional data sources are often desirable.

4.2.1.1 Base-object auditing

One method tracking system data that was considered, but not implemented is base-object auditing. Windows NT/2000 provides a level of base-object auditing that allows users to audit access to objects. Within the Local Polices inside Local Security Settings, a user can turn on several auditing options, including object access and system events. When enabled, base-object auditing can audit access success, failure, or both to objects and events. One shortcoming of this approach is that the auditing method does not provide desirable information such as the thread that triggered an event. Since essentially flipping an on-off switch controls base-object auditing, the information available is all or nothing without any way to modify the data available.

4.2.1.2 Strace for NT

The method of system call tracking implemented for AppID was *strace for NT*, a utility for NT system calls made by a process based on the *strace* on Linux and other Unix operating system [11]. One advantage over base-object auditing is that *strace* provides an open-source implementation. Because it's open-source, we are able to control what *strace* audits. When an application the user is interested in is launched, *strace* provides a way to control and log what system calls an application is allowed to use. The *strace* utility uses a hooking method performed by a device driver that hooks all system calls and checks for the SeDebugPrivilege before allowing the application to open it. The

device driver responsible for the hooking also collects data that in turn can be utilized by auditing components within AppID.

4.2.2 Architecture

4.2.2.1 Pipe-line design

AppID is a pipeline-based system with data being collected at a source, processed through various components, and finally sent to a data sink where important information is reported to the user. Using a pipeline architecture means that AppID is extensible and modifiable. For example, components can be written and inserted into the pipeline to perform a new functions on the source data. Specifically, the intrusion detection algorithms employed by AppID can be easily changed or complemented by additional algorithms. The figure below shows the pipeline utilized for NT-based systems for auditing system access.

```
+-----------------------------------------------------------+
|                                                           |
|  +-----------+    +-----------+    +----------+    +----------+  |
|  |           |    |           |    | NGram    |    | Report   |  |
|  | NTCollector| => | NTtoID    | => | Algorithm| => | Manager  |  |
|  |           |    |           |    |          |    |          |  |
|  +-----------+    +-----------+    +----------+    +----------+  |
|                                               /      \          |
|                                              /        \         |
|                                    +-----------+  +-----------+ |
|                                    | Log File  |  | Event Log | |
|                                    | Destination| | Destination||
|                                    +-----------+  +-----------+ |
|                                                           |
+-----------------------------------------------------------+
```
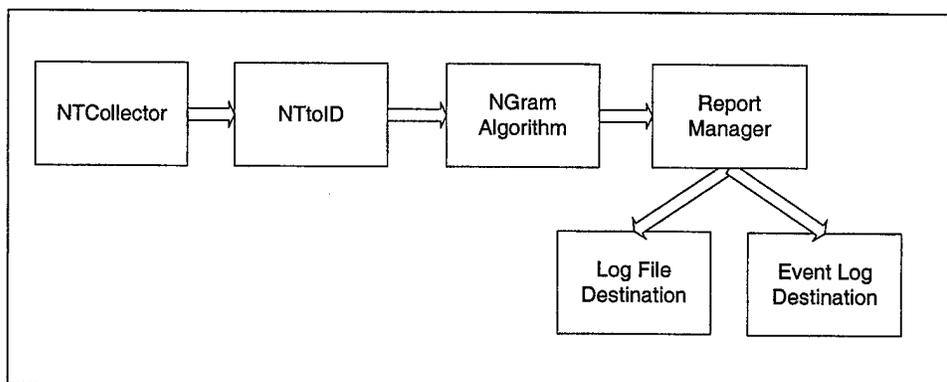
4.2.2.2 Components

The two principle components of the pipeline are *sources* and *sinks*. Additional components perform various tasks such as parsing data and performing anomaly analysis. Sources are the input for the pipeline, and are responsible for taking data from wherever it is stored and feeding it to the pipeline for analysis. The source implemented for AppID is an NTCollector that works in conjunction with the Strace device driver. This source takes data from Strace, and stores it in memory until it is processed thru the pipeline. Sinks are the final step in the pipeline, and are used to process data and transform or convert the data into its final format. As an example, a sink will receive the data that has been produced by the other components of the pipeline and write messages to the Windows Event Log where the data can be used by other applications. Several sinks are available in AppID and are determined by the mode (e.g., data collection or recall) in which the user is running AppID.

The components in the pipeline are specified using a configuration file. The first component in the system must be a source, and as indicated above, AppID utilizes an NTCollector source. As shown in the sample pipeline above, the next component for the NT based system is a component that converts NT type data to a generic type of data known as IDData. This component also adds various NT-specific data to the IDData

data, such as *process name*. The next component is a Configuration component, which sets up all the proper sub-components for using the NGram Algorithm. After data has passed through the NGram algorithm, it is sent directly to the sink, which in this case is Report Manager.

4.2.2.3 Algorithms

For purposes of testing the AppID framework we implemented the relatively simple and well-known *stide* algorithm[8]. This algorithm works by sliding a window of fixed size over the normal behavior data, which consists of sequences of system calls. Each series of n consecutive system calls is recorded in a database. When the system is used in the field, any sequence of six system calls leads to an alarm if it is not already in the database. This approach is based on the assumption that all normal sequences were seen and recorded during training. A weakness of this approach is that it can detect n-grams that have never occurred during the normal behavior of an application, but it cannot detect n-grams that have occurred in some contexts but not in others.

Prior work in intrusion detection has shown this to be the most effective means of detecting anomalies with the least amount of false positives when the training data set is large [9]. In this paper, the authors achieved best results for the n-gram algorithm when n had the value three. This yielded detection rates well above 90% with false positives approaching less than 2%.

5 Conclusions and Future Work

The technologies developed for this contract address two of today's critical security issues. The BayeScan prototype and Malicious Email Filter provide end-users with access to new techniques for identifying novel malicious software before it has a chance to damage a system. These prototypes are based on leading research into using data mining techniques for identifying malicious programs. The resulting tools extend existing antivirus solutions with additional detection capabilities. Appendix B contains a complete guide to installing and using BayeScan.

The AppID framework facilitates future research and development of real-time host-based intrusion detection technologies. At Cigital we have developed a number of promising intrusion detection techniques that would benefit from the AppID framework. At this time we have only fitted AppID with the simple *stide* algorithm. This enabled functional evaluation of AppID, but does not produce very strong intrusion detection results. Future work would include incorporating state-of-the-art intrusion detection algorithms into the AppID framework. Appendix C contains a guide for installing and using AppID, along with instructions for developing new AppID components.

6 References

1. The WildList Organization International. http://www.wildlist.org (June 10, 2002).

2. M. Schultz, E. Eskin, E. Zadok, and S. Stolfo. "Data Mining Methods for Detection of New Malicious Executables," in *2001 IEEE Symposium on Security and Privacy*, May 2001.

3. B. P. C. Warrender, S. Forrest, "Detecting intrusions using system calls: Alternative data models," in *1999 IEEE Symposium on Security and Privacy*, pp. 133-145, 1999.

4. A.P. Kosoresow and S. A. Hofmeyr, "Intrusion detection via system call traces," *IEEE Software*, vol. 14, pp. 24-42, Sept./Oct. 1997.

5. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *2000 IEEE Symposium on Security and Privacy*, pp. 144-155, IEEE Computer Society, 2000.

6. R. Sekar and P. Uppuluri, "Synthesizing fast intrusion prevention/detection systems from high-level specifications," in *Proceedings of the 8th USENIX Security Symposium* (SECURITY-99), (Berkely, CA), pp. 63-78, Usenix Association, Aug. 23-26 1999.

7. M. Schultz, E. Eskin, and S. Stolfo. "MEF: Malicious Email Filter - A UNIX Mail Filter that Detects Malicious Windows Executables," in *USENIX Annual Technical Conference - FREENIX Track*, Boston, MA, June 2001.

8. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for Unix processes," in *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pp. 120-128, IEEE Computer Society, IEEE Computer Society Press, May 1996.

9. C. C. Michael and A. Ghosh, "Two state-based approaches to program-based anomaly detection," in Proceedings of ACSAC 2000, pp. 21-30, December 2000.

10. C. C. Michael and A. Ghosh, "Simple state-based approaches to program-based anomaly detection." To appear in ACM TISSEC, August 2002.

11. Strace for NT readme file. http://razor.bindview.com/tools/desc/strace_readme.html (June 10, 2002).

12. A.K. Ghosh, A. Schwartzbard and M. Schatz, "Learning Program Behavior Profiles for Intrusion Detection," *1st USENIX Workshop on Intrusion Detection and Network Monitoring*, 1999.

13. K. Nigam, A. McCallum, S. Thrun, and T. Mitchell. "Learning to classify text from labeled and unlabeled documents," in Proceedings of the 15th National Conference on Artificial Intelligence, AAAI-98, 1998.

Appendix A: Test framework configuration file options

This section explains the available configuration options for working with the malicious software detection framework.

- **command** – the **command** option determines whether the *analysis* program will be executed in the default *cross-validation* mode, or in *interactive* mode. Setting the **command** option to "generate" causes the program to build a classification model using the full data set of benign and malicious programs. If the **command** option is not specified, the program uses cross-validation, dividing the data sets into five equal sets, four of which will be used to build the classification model for classifying the remaining set.
- **file** – the **file** option is used in conjunction with the "generate" **command** option to specify the name of the file to be created to store the classification model.
- **unique** – the **unique** option is a flag that is set to "true" or "false", with "false" as the default value used when the **unique** option is not specified.
- **malicious** – the malicious option is used to specify the list file (.list) containing the list of viruses to be used in the test, both for training (building the model) and testing
- **benign** – the **benign** option is used to specify the list file (.list) containing the list of benign files to be used in the test, both for training and testing.
- **increment** – the **increment** option is used to specify the step increment to be used when selecting files for training and testing sets. By default the **increment** is 1, but a different integer value can be specified to reduce the size of the data sets used for the test.
- **include_train** – the **include_train** option is a flag that is set to "true" or "false", with "false" as the default value used when the **include_train** option is not specified for a *Date-Ordered* test (see test option "3" below). If set to "true" the include_train flag causes the training set for a Date-Ordered test to include the training files in the test set.
- **input_directory** – the input_directory option is used to specify the location of the intermediate listing files containing the testing and training file lists. This option is used only for test option "6" (see test option below).
- **output_directory** – the **output directory** option is used to specify the directory location to be used by the analysis program to store all output files. The output files contain the results of the various tests under each algorithm (classifier).
- **write_model** – the write_model option is a flag that is set to "true" or "false", with "false" as the default value used when the **write_model** option is not specified. If the flag is set to "true", the classification model, containing a list of features along with the corresponding number of times they occur in benign files, and the number of times they occur in malicious files, is written to a file
- **min_size** – the **min_size** option specifies the lower bound on the size of features to be used for classifying each file
- **max_size** – the **max_size** option specifies the upper bound on the size of features to be used for classifying each file
- **use_reducing** – the **use_reducing** option is a flag that is set to "true" or "false", with "false" as the default value used when the **use_reducing** option is not specified.
- **use_size** – the **use_size** option is a flag that is set to "true" or "false", with "false" as the default value used when the **use_size** option is not specified.
- **single_test** – the **single_test** option is a flag that is set to "true" or "false", with "false" as the default value used when the **single_test** option is not specified.
- **test** – the **test** option specifies the test to be executed by the *analysis* program.
  - "1" – executes the *Sample* algorithm using cross-validation
  - "2" – executes a Random sampling procedure to select training and test data sets from the available data set for a cross-validation test

- "3" – executes a *Date-Ordered* test. A training set of malicious files is selected based on the date associated with the virus executable. The set of benign files for the model is selected using a random sampling procedure. The remaining virus files are tested in a date order, earlier files first, with the tested file assimilated into the classification model afterwards.
- "4" – executes a *Reversed Date-Ordered* test. The procedure used above for test "3" is used, but all ordered file selections are done by selecting the most recent files from the data set first.
- "5" – executes a Random sampling short test.
- "6" – executes a test using the same classification models as the previous execution. A single_test can also be specified for this test – see single_test option below.

Appendix B: *BayeScan* User's Guide

1. Installing BayeScan

An InstallShield wizard is provided for installing BayeScan on a user's desktop. To install BayeScan, simply double-click the setup executable (setup.exe) in the BayeScan folder on the CD, or run setup.exe from the Start Menu's Run command.
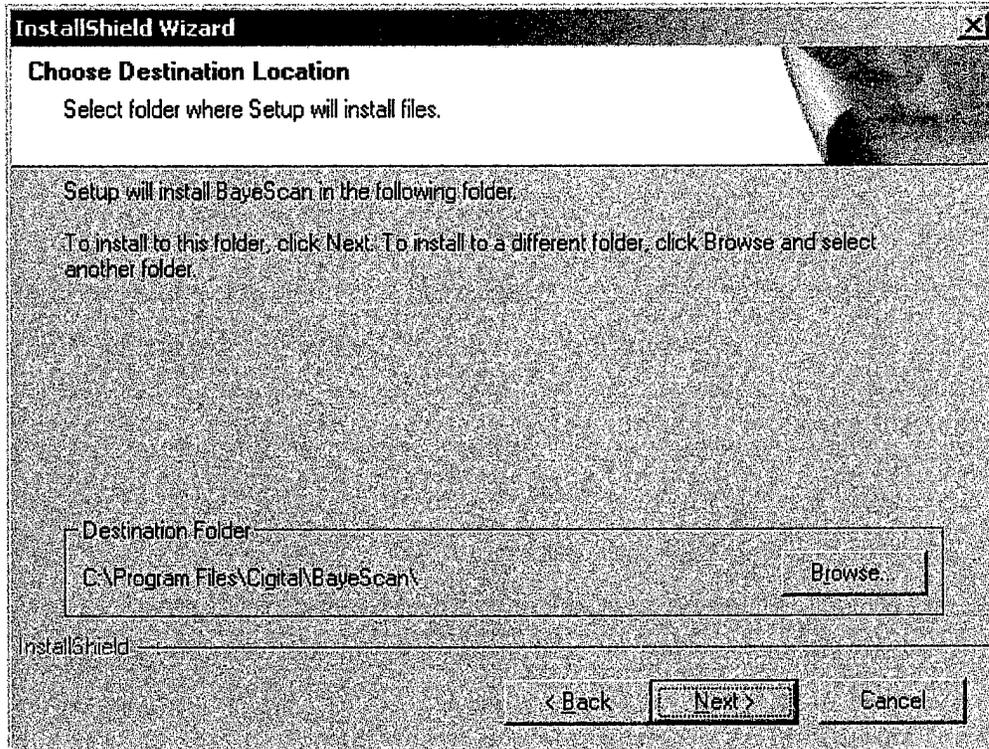
Once the installation program begins, the user is asked to specify the directory in which to install BayeScan. To install BayeScan in a location other than the default directory, click the Browse button and either browse to, or enter, the desired location. After verifying that the Destination Folder is correct, click the Next button to begin the install process. During the installation, the status of the install is displayed. The user is notified upon completion of the install.

2. Using BayeScan

To start BayeScan, double-click the executable (BayeScan.exe) from inside Windows Explorer.

2.1. Selecting a Classification Model

The *Model/Select* menu item is used to select the classification model for BayeScan to use. To verify the current model or change to a different model, choose *Select* from the *Model* menu dropdown.
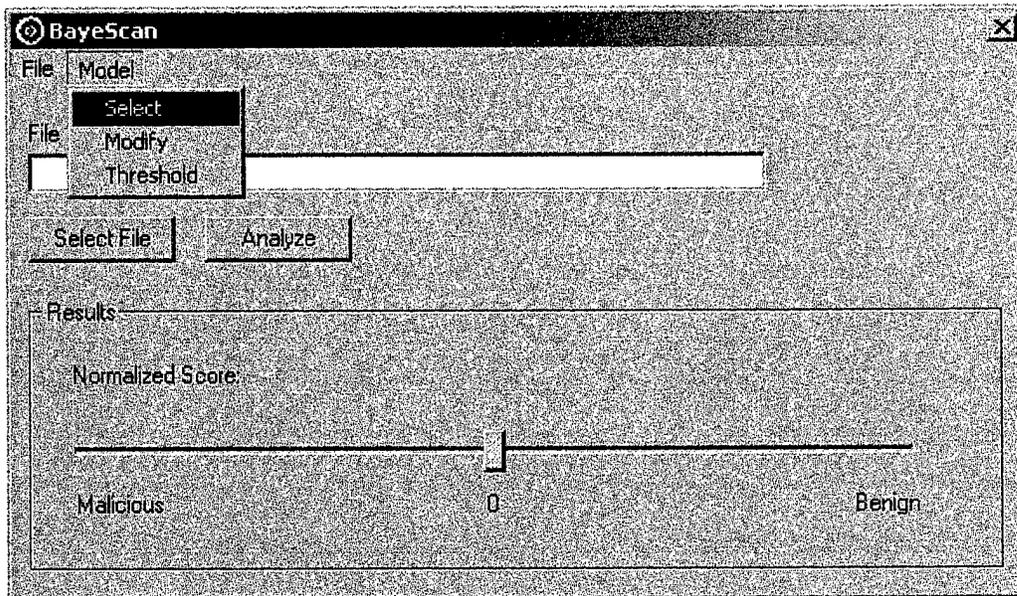
The *Model/Select* menu option opens the Model Selection window shown below.
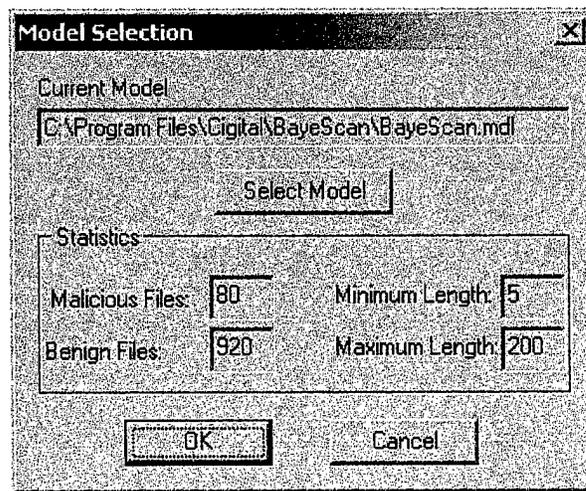
The Model Selection window displays data pertaining to the model – the number of malicious files and benign files used to construct the model, as well as the minimum and maximum string length for strings in the model. If the current model listed in the Model Selection window is the model you want to use/modify, click OK. To select a different model, click the Select Model button and browse to find and select the desired model. Click OK.

2.2. Analyzing an Executable

BayeScan provides a user the ability to analyze the maliciousness of a new un-trusted executable. To analyze a new file, click the Select File button and select the un-trusted file.
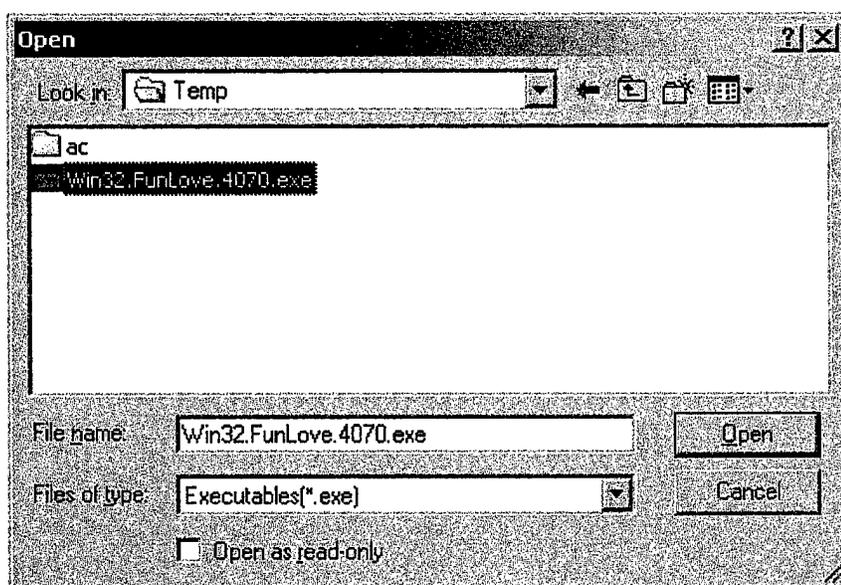
After identifying the file to classify, click the Analyze button.

BayeScan will then assign a score for the maliciousness of the file. A negative score indicates that the file is classified as malicious, and a positive score indicates that the file is classified as benign. The lower the score, the more likely an executable is malicious. For instance, a file with a score of –500 is much more likely to be malicious than a file with a score of –10. Likewise, a file with a score of 1000 is more likely to be benign that a file with a score of 50. The variation in the scores is predicated on the number of matches found when comparing the strings in the analyzed executable to the strings in the classification model.
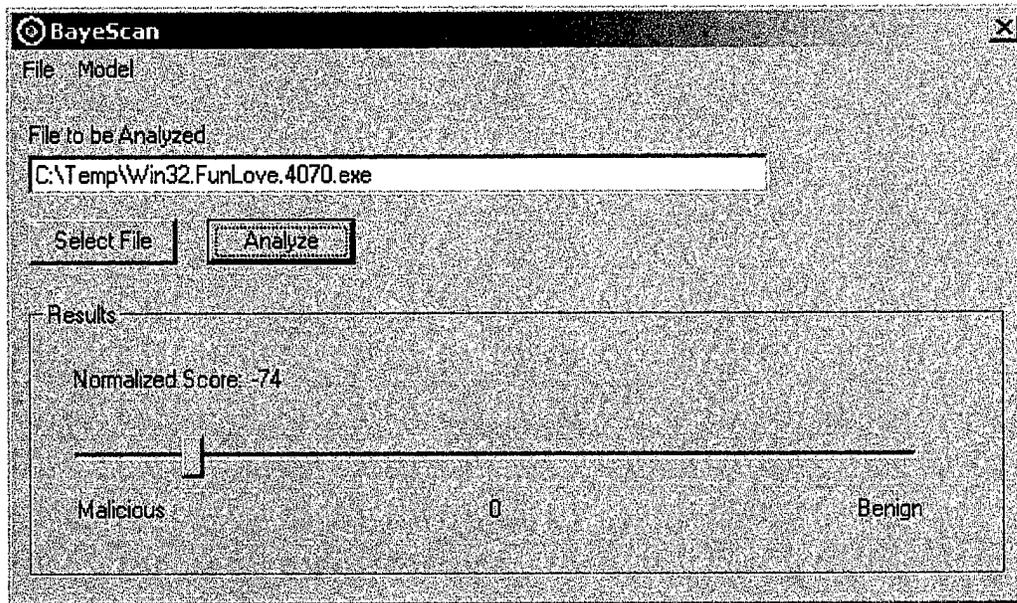
2.3. Modifying the Current Model

The *Model/Modify* menu item is used to update the current classification model being used by BayeScan. To update the current model, select the *Modify* option from the *Model* menu dropdown.
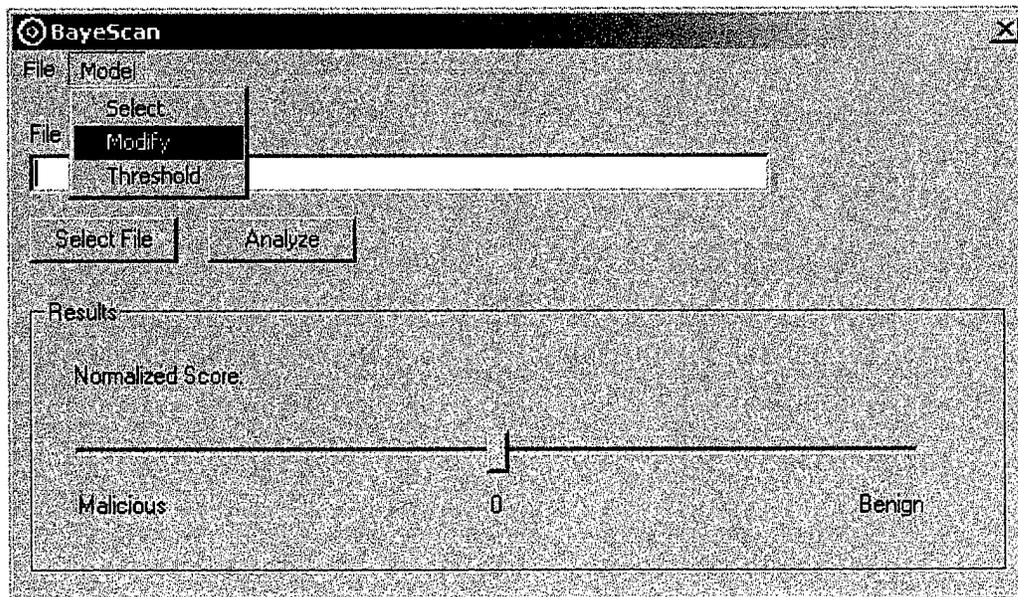
After selecting the *Modify* option, the Model Modification window is displayed.
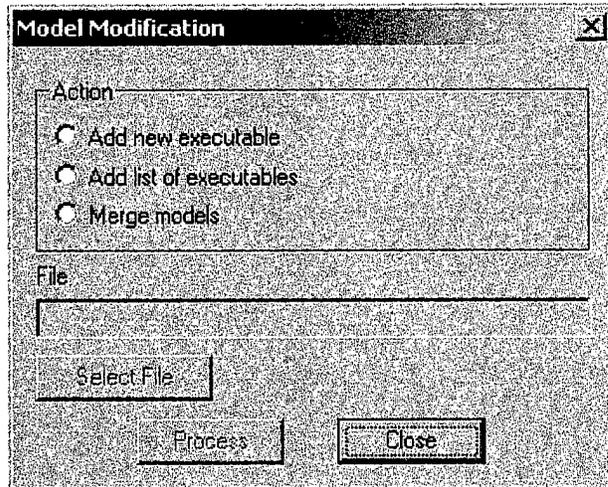
The classification model can be modified in three ways: add a new executable to the model; add multiple executables to the model; merge a different model to the existing model.

2.3.1. Adding a New Executable to the Model

BayeScan allows the user to add new executables to the existing classification model. To add a new executable to the model, first select "Add new executable" from the Model Modification menu.
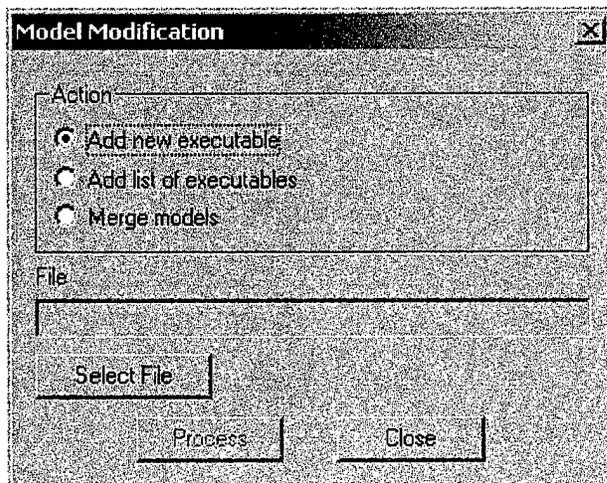
Next, click the Select File button and select the executable (.exe) file to add to the model. In the example below, the ScriptGuard executable, ScriptGuard.exe, is selected.
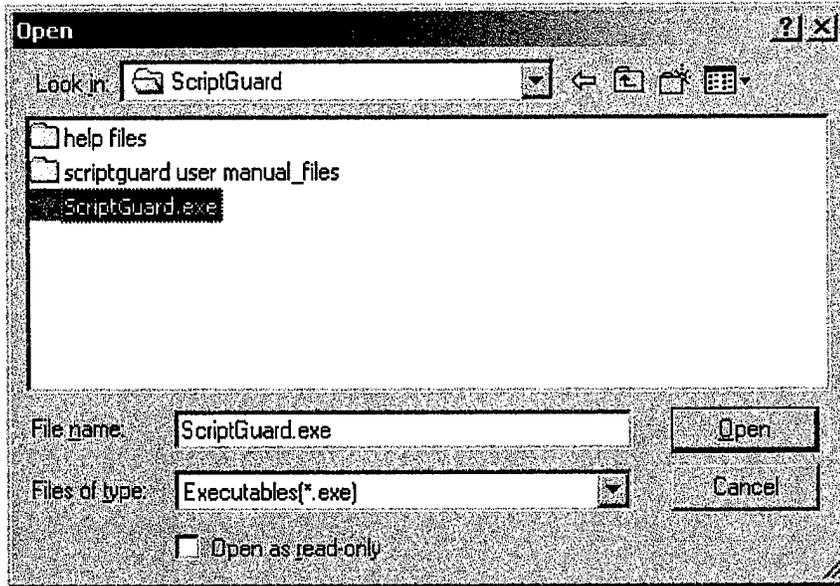
Once the file is selected, click the Process button.
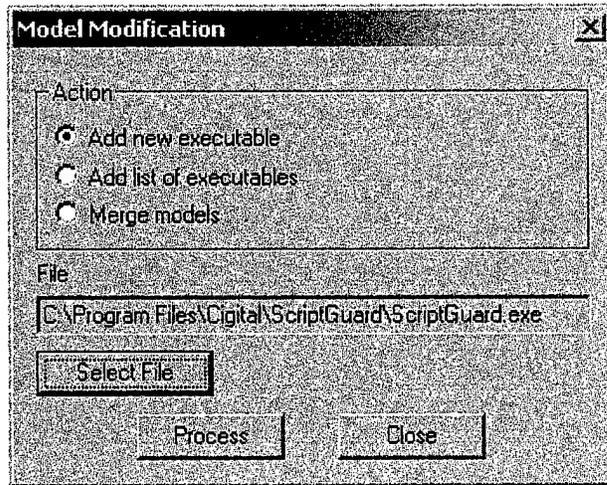
The user must then tell BayeScan if the executable file is a malicious or benign. If the executable is malicious click Yes, and otherwise click No.
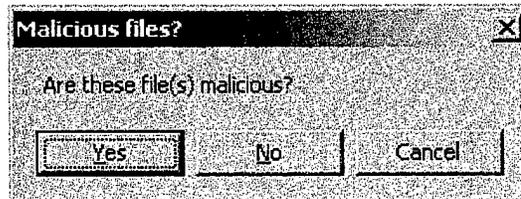
A Results window pops up after the new file is processed, and indicates if the new executable has been added to the model. If the file already exists as part of the model, the file will not be added to the model and the Results window will say "Added 0 file(s)".
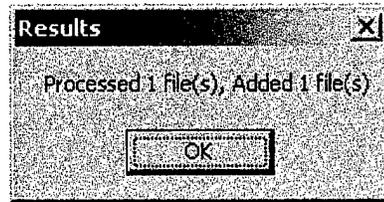
2.3.2. Adding Multiple Executables to the Model

BayeScan allows the user to add more than one executable at a time to the existing classification model by creating a list (.lst) file with the full path name of the executables to add to the model. Below is an example of a list file with three files.

C:\WINNT\discover.exe
C:\WINNT\explorer.exe
C:\WINNT\system32\cleanmgr.exe

Note that the list must contain only one type of file – malicious or benign. Once the list file has been created, select "Add list of executables" from the Model Modification menu, and click on the "Select File" button to select the name of the list file.
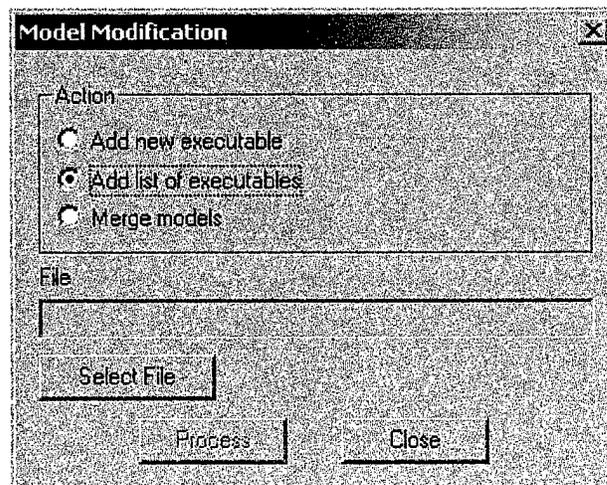
Like the process for adding a single executable to the model, the user must tell BayeScan whether or not the files in the list are malicious. Depending on the number of files in the list and the size of the executable files, this process may take several minutes to complete. Upon completion, a Results window will be displayed showing the number of files processed and the number of files added to the model (remember that only new files are added to the model).

Results

Processed 1368 file(s), Added 919 file(s)

OK
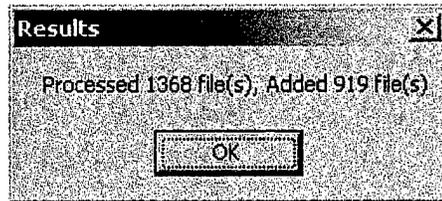
2.3.3. Merging Model Files

BayeScan allows the user to combine two distinct classification models to form a new updated model. To merge a new model to the current model, first select "Merge models" from the Model Modification menu and click the Select File button. After selecting the new model (.mdl) file, click the Process button.

Model Modification

Action
- Add new executable
- Add list of executables
- Merge models

File

C:\Program Files\Digital\BayeScan\update.mdl

Select File

Process   Close
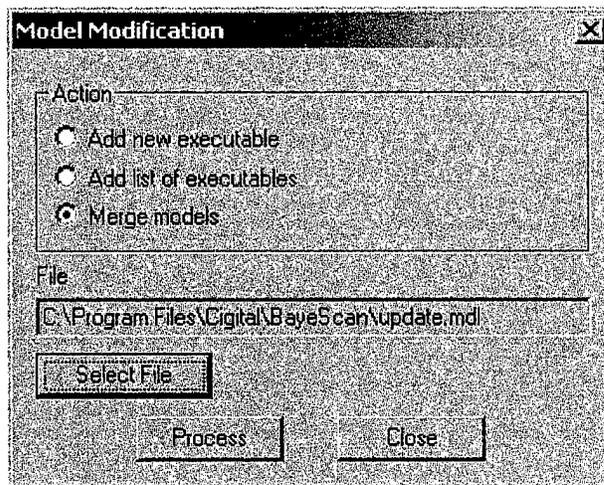
Before merging the two models, BayeScan will analyze the two models and if an overlap is found between the two models – one or more files used to create the models appears in both models – an Overlap Present message appears.

Overlap present

Overlap present between models.
35 Files in common between models
Continue with merge?

Yes   No
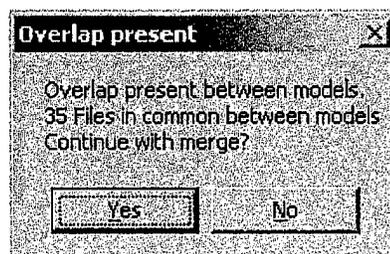
If you elect to merge the two models, a Results window will pop up at the completion of the merge summarizing the changes made to the current model.
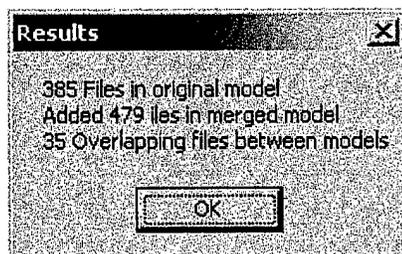
```
Results                          X
385 Files in original model
Added 479 les in merged model
35 Overlapping files between models

              OK
```

The Merge Models functionality is designed to merge distinct models. For instance, if a model is built for a new class of files that were previously not in the original model, the merge functionality is an effective way to expand the range of protection provided by BayeScan. Merge Models is also the method used for updating the model for a new group of viruses or benign executables that have been analyzed to form a new model, effectively a "BayeScan update" model.

2.4. Changing the Classification Threshold

BayeScan allows the user to change the threshold used for classification. The threshold is a means of fine-tuning the BayeScan classifier by adding (or subtracting in the case of a negative threshold) a standardized value to the analysis score when deciding if a new executable is malicious. A threshold of 150 is the default threshold for BayeScan.

To change the threshold, select *Threshold* from the *Model* dropdown menu.
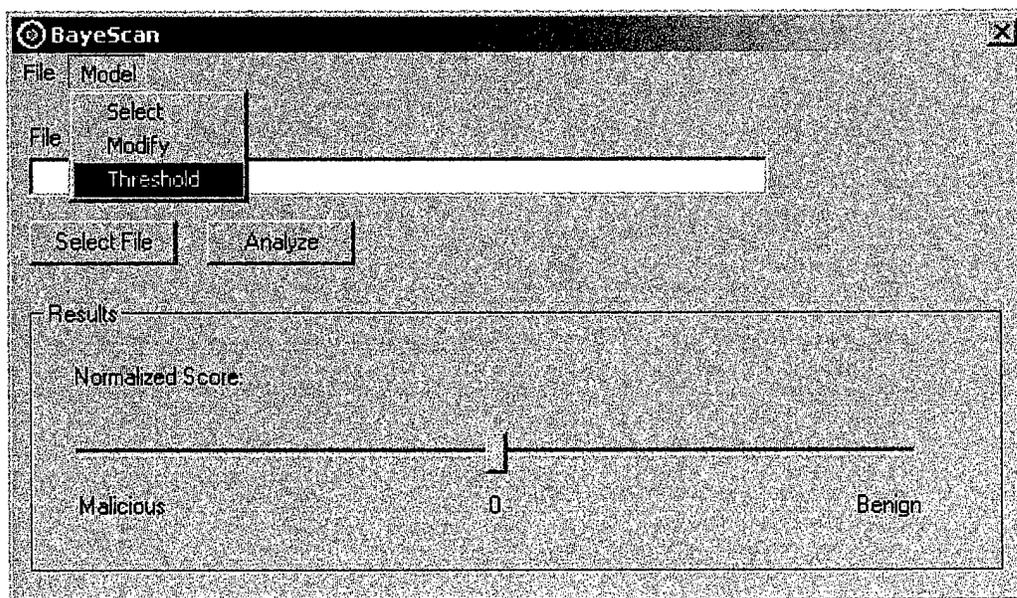
The Model Threshold window displays the current threshold, and allows the user to select a preset threshold – Low, Medium, or High – or enter a new Custom Threshold. To use a preset threshold, simply select the desired value. Otherwise, select Custom Threshold, enter the new threshold value, and click the OK button.
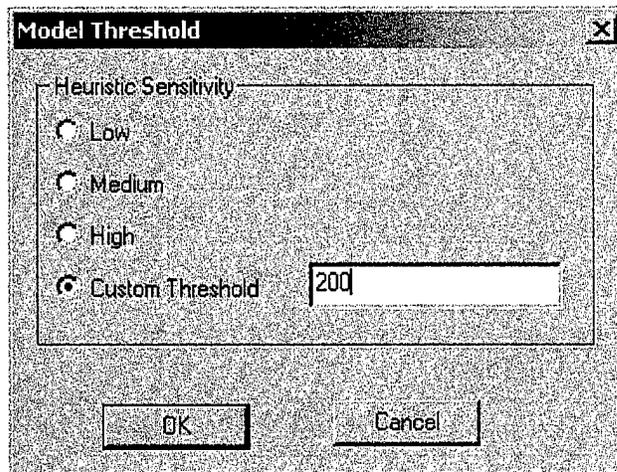
The new threshold value will then be used for all BayeScan classifications until the user sets a different threshold.

2.5. Exiting the Application

In order to exit the application, the user must use the *Exit* item in the *File* menu dropdown.
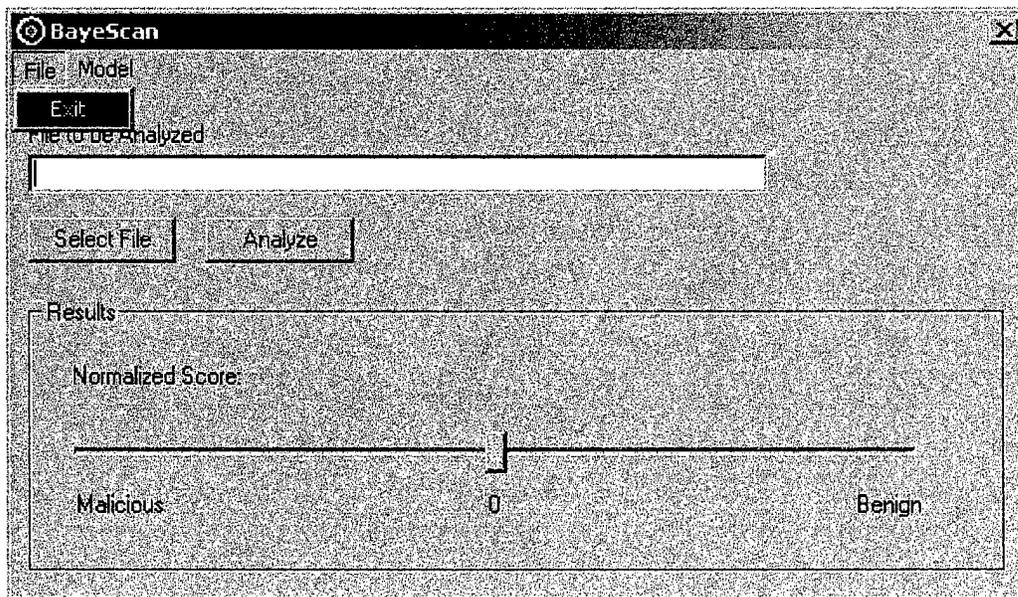
Closing the BayeScan window by clicking the 'X' on the title bar simply hides the window, and does not actually end the program. To reopen the BayeScan user window, just click on the application's icon in the Windows system tray.

Appendix C: AppID User's Guide

Introduction

AppId (Application Intrusion Detection) was developed to provide real-time detection of intrusions by comparing current program behaviors to a set of known "normal" behaviors. AppId works by collecting data on a set of applications during normal usage. AppId is then put into a training mode where it builds a model of normal behavior based upon the collected data. Following training, AppId is put into a recall mode where it compares the current behavior for these applications against the normal behavior model and reports any anomalous behavior as possible intrusions.

The overall architecture of AppId is a data pipeline. System data is fed into a source component at the beginning of the pipeline. It is processed using various components along the way, and at the end of the pipeline a report detailing a detected intrusion is sent to the user. All of the pipeline components inherit from generic component classes. This allows those components to be replaced transparently without requiring recoding of the rest of the pipeline. This would allow new intrusion detection algorithms to be introduced into the system. There is a pipeline-manager that assembles the pipeline based upon a configuration file, and handles the passing of data between pipeline components.

Installation

When the AppId installation zip file is expanded, an AppId install folder will be created (see Figure 1). Double click the setup.exe file to begin the install.

Figure 1 – AppId Install Folder

The InstallShield program will being executing and present the user with an initial welcome screen (see Figure 2) after some operating system information has been collected.
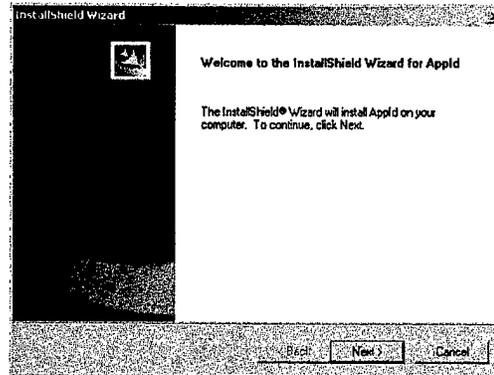
Figure 2 – AppId InstallShield

Selecting the Next button will send the user to the location selection dialog (see Figure 3). The user may select an install location for AppId. A default location is suggested, but any location is acceptable. Once the location has been selected, the user should select the Next button.
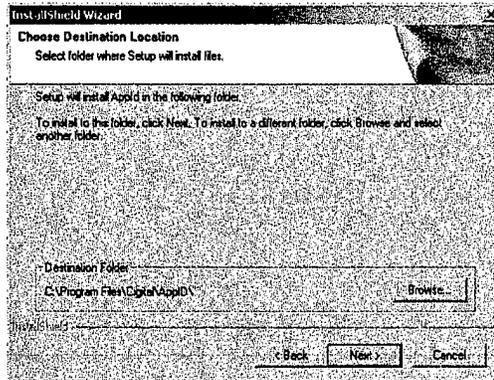
Figure 3- Location Selection Dialog

Once the installation is complete, InstallShield will display the installation complete dialog (see Figure 4). Once InstallShield has completed, the user must reboot the system to begin using AppId.
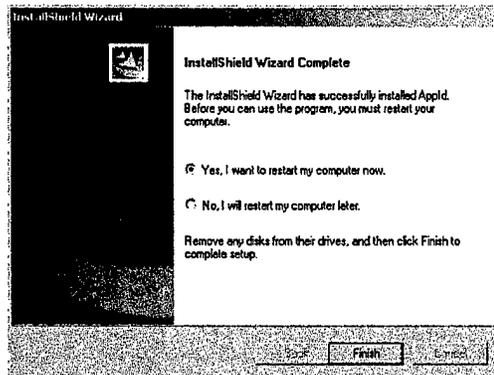
Figure 4 – Installation Complete Dialog

Using the Program

AppId Control Panel

The main interface to AppId is the AppId control panel. To access the control panel, select the Start Menu in the lower left corner of the windows desktop, then select Settings, then Control Panel. It will bring up a window similar to Figure 5. Double click the icon labeled AppId to access the AppId system.
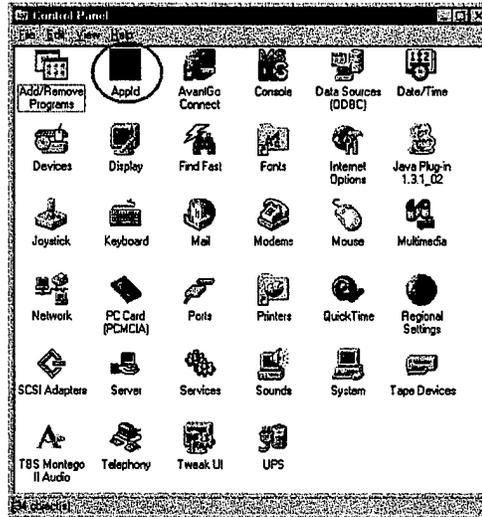
Figure 5 – Windows Control Panels

The AppId control panel (see Figure 6) allows the user to perform mode selection (data collection, training, and recall), start and stop of the service, alter the configuration files, and modify the current application set.
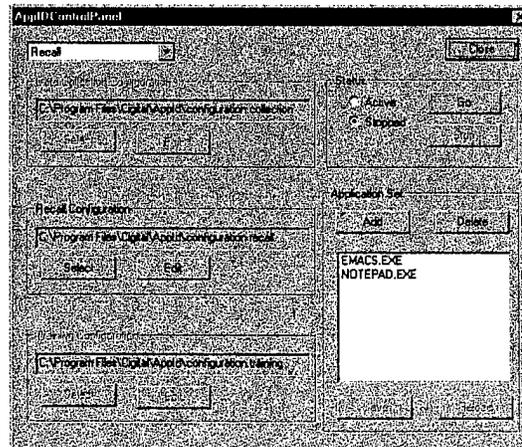
Figure 6 – AppId Control Panel

Mode

The mode can be selected by using the drop down menu on the AppId Control Panel (see Figure 7). This menu will only be activated if the system is in the stopped state. Selecting one of the modes will cause the configuration for that mode to become active (see Figure 8). The mode that is currently selected will be the default startup mode.
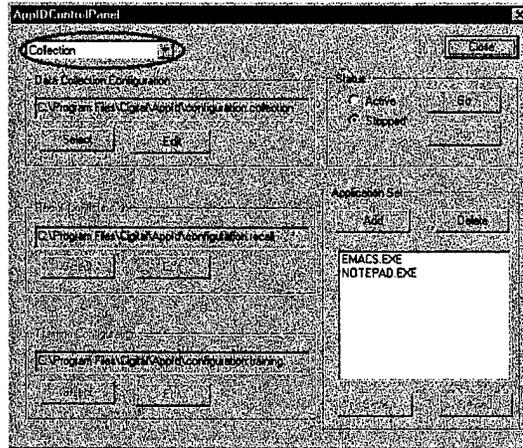
Figure 7 – Mode Selection Menu
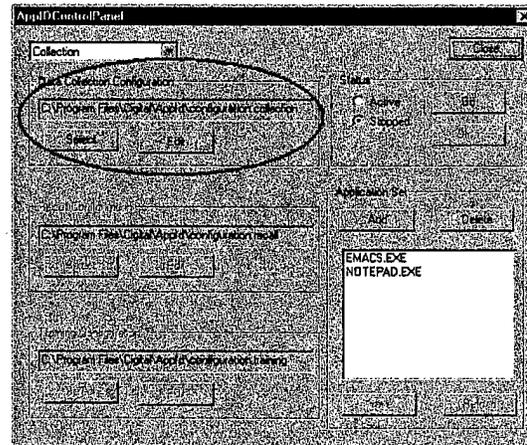
Figure 8 – Data Collection Configuration

AppId can be run in one of three modes (Data collection, Training, and Recall). Data collection mode allows AppId to collect information on applications that are specified in the current application set. An application set is a list of applications that the user is interested in monitoring for intrusions. When the system is running in data collection mode and an application in the current application set is launched, AppId will begin to collect data on how the application interacts with the operating system. The data collected will constitute "normal" behavior for the application. The user should use the applications during data collection mode as thoroughly as possible in the way they anticipate them being used during the recall mode. This is important so that AppId can get a representation of the normal behavior for the application. For example, if the user wants to monitor a web server and plans to deploy java server pages soon, then AppId must watch the system after the java server pages have been deployed or their deployment will appear to deviate from "normal" behavior and be seen as an anomaly and therefore, an intrusion.

Once the user has gather sufficient data, the system can be trained and a model of "normal" behavior built. The system can then be put into recall mode. AppId will begin to compare the current behavior of the system to the "normal" behavior that has been trained into the model. The user will be informed when the current behavior deviates from the "normal" behavior.

Figure 9 shows the recall configuration available after the recall mode has been selected. The user can use the configuration area to select a new configuration file or to edit the current configuration file for the current mode. Pressing the Select button (see Figure 10) for any of the configurations will bring up a file selection box (see Figure 11). Pressing the Edit button (see Figure 12) will bring the current configuration file up inside of Notepad so that it can be altered.
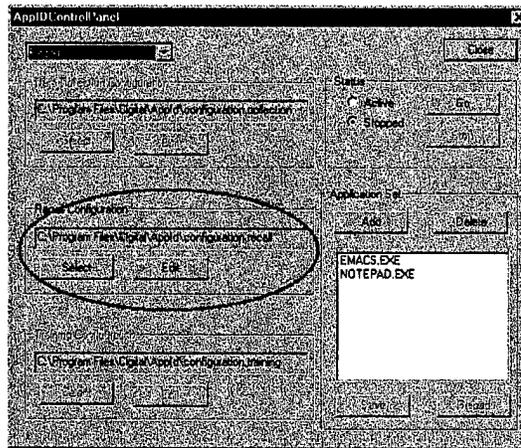
Figure 9 – Recall Mode Selected
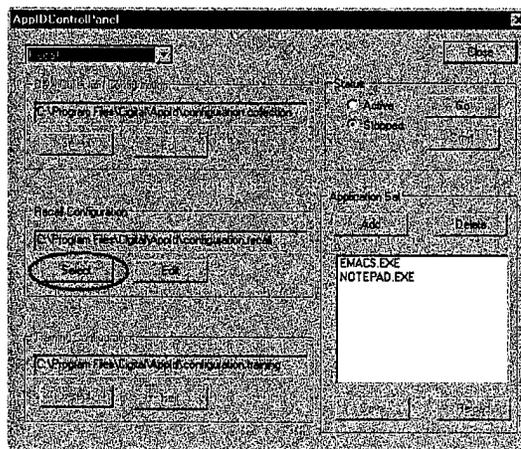
Figure 10 – Select Button
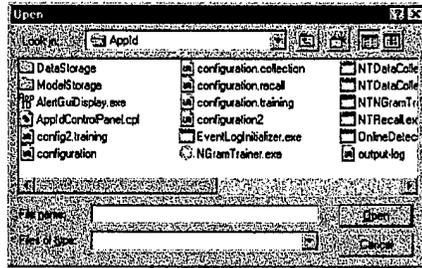
Figure 11 – File Selection Box
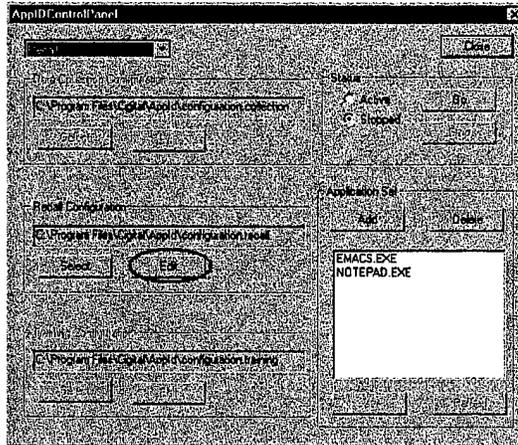
Figure 12 – Edit Button
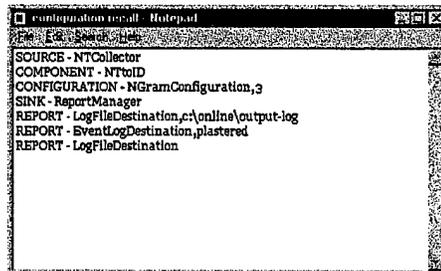
Figure 13 – Notepad being used to edit a configuration file

Status

The Control Panel also shows the current status of AppId (see Figure 14). The system can either be active (i.e. running) or stopped. If the system is active, it will remain active upon reboot (i.e. the system will automatically come up in the active status). If the system is stopped, it will not run automatically upon reboot.
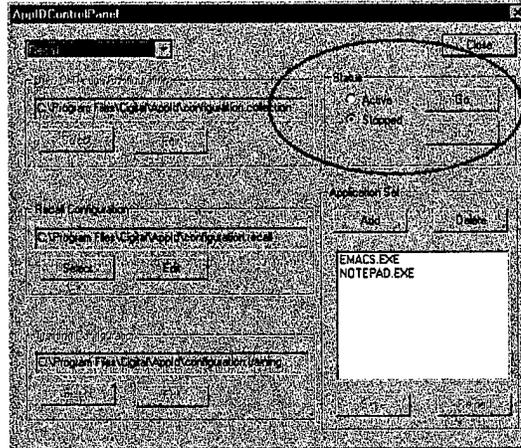

Figure 14 – Status Area of the AppId Control Panel

Selecting the Go button (see Figure 15) while the system is stopped will start the system running in whatever mode is currently selected in the mode selection menu. Once the system has started, the AppId Control Panel will update the status to active (see Figure 16). The Stop button will now be enabled and can be selected to stop the system.

Figure 15 – Go Button

Figure 16 – AppId in Active State

If the user selects the Go button while in training mode (see Figure 17), the program will immediately run the training manager program in another window (see Figure 18). This operation cannot be stopped or canceled.

Figure 17 – Training Mode Selected

Figure 18 – Active Training Mode
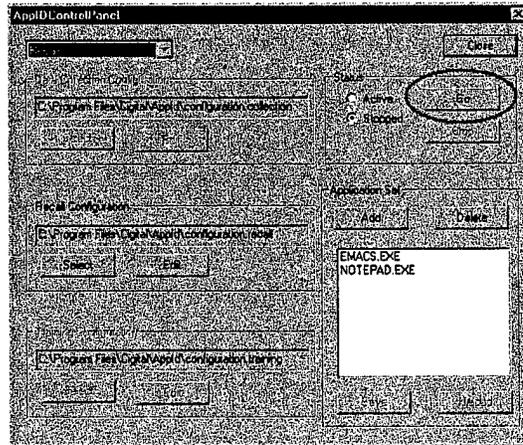
Application Set

AppId can monitor a user-specified set of applications. The current applications being monitored are listed in the application set portion of the control panel (see Figure 19). This list can be modified by using the Add or Delete buttons found in the application set portion of the control panel. Selecting the Add button (see Figure 20) will bring up a file selection dialog box (see Figure 21). Since AppId can only monitor executable programs, only executable programs (i.e. ending in .exe) will be shown to the user in the file dialog box. Adding or removing an application from the list will activate the Save and Reload buttons (see Figure 22). The Save button will save the current altered application set while the Reload button will return the previously saved application set. To delete an application from the list, select it in the list and then press the Delete button.
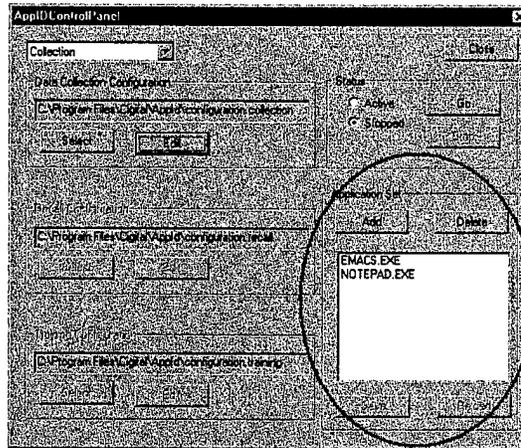
Figure 19 - Application Set Portion of the control panel
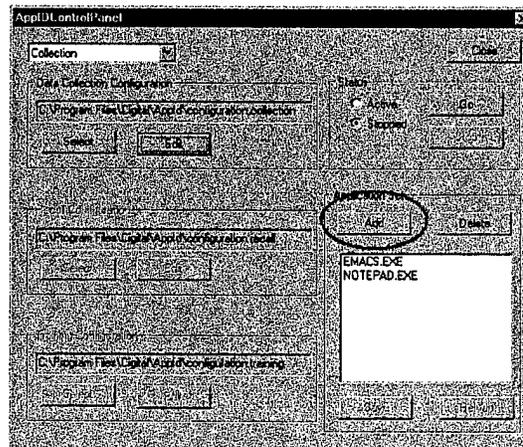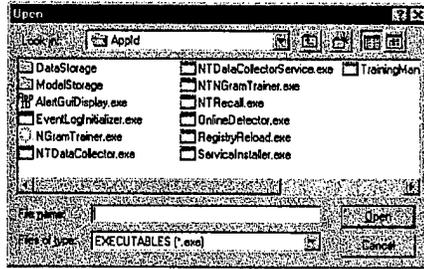

Figure 20 - Add Button

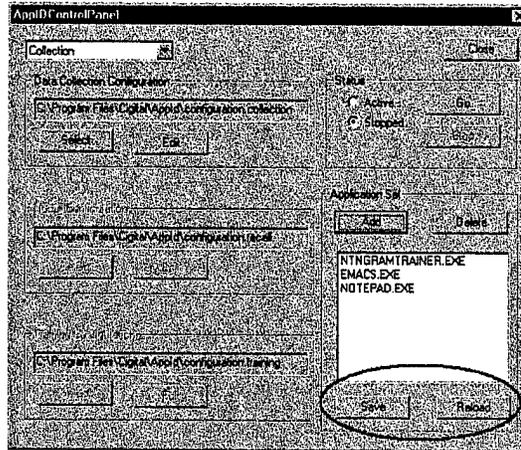Figure 21 – File selection box to add applications to watch

Figure 22 – Enabled Save and Reload buttons

Configuration Files

Each mode of AppId has a related configuration file. These configuration files allow AppId to be configured dynamically at runtime. The configuration file specifies the order and type of components within AppId. AppId is a pipeline-based system with data being collected at a source, processed through various components, and finally sent to a data sink where important information is reported to the user. Each component is built to take inputs and to produce outputs. The components must be linked together in the correct order or the system will not function. An example configuration file for Recall mode is shown in Figure 23.

```
SOURCE - NTCollector
COMPONENT - NTtoID
CONFIGURATION - NGramConfiguration,3
SINK - ReportManager
REPORT - LogFileDestination,C:\Program Files\Cigital\AppId\output-log
REPORT - EventLogDestination
```

Figure 23 – Sample Configuration File for Recall Mode

The first component in the system must be a source and, for NT-based systems, must be an NTCollector source. The next component for NT-based systems is a simple component that converts NT type data to a generic type of data known as IDData. This component also adds various NT-specific data to the IDData data, such as process name. The next component is a Configuration component, which sets up all the proper sub-components for using the NGram Algorithm. After data has passed through the NGram algorithm, it is sent directly to the Report Manager. The Report Manager can have any number of report destinations. These report destinations handle the report in the most appropriate manner for the destination. For example, writing to the windows event log requires special commands that are embedded within the event log destination component. The pipeline built by the configuration file shown in Figure 23 can be seen in Figure 24.
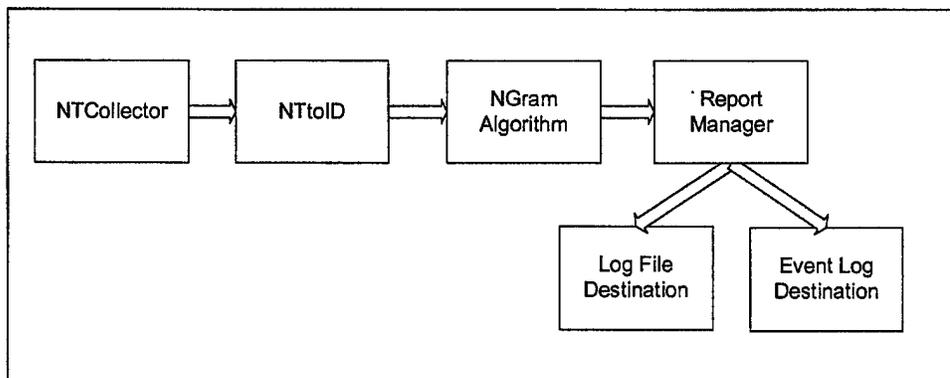

Figure 24 – Pipeline for Recall

An example configuration file in the same type of format for data collection is shown in Figure 25. The pipeline for data collection is very similar in terms of getting the data from the operating system. The source of the data is the NTCollector. The data is passed to the NTtoID component for conversion to IDData. The data is then sent to the Process Splitter, a sink that writes the information out to data files. These data files will be used during training to build the model of normal behavior. The pipeline built by the configuration file shown in Figure 25 can be seen in Figure 26.

```
SOURCE - NTCollector
COMPONENT - NTtoID
SINK - ProcessSplitter
```

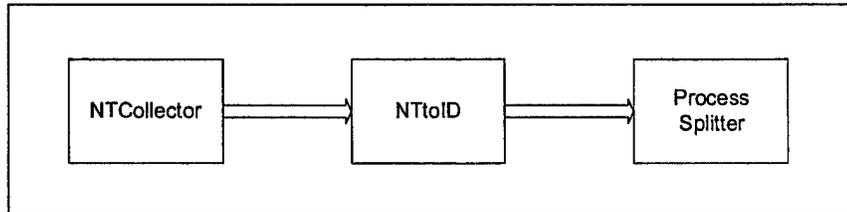Figure 25 – Sample Configuration File for Collection

Figure 26 – Pipeline for Data Collection

The configuration file for training will be manipulated more often that the recall or data collection configuration files. A sample configuration file for training is shown in Figure 27. The first line of the file is the name of the training executable to use. Since only one algorithm is distributed with the system, the training executable must be NTNGramTrainer.exe. The remaining lines in the configuration file are command line arguments for the training executable. NTNGramTrainer executable takes the size of the NGram window as its first argument. The remaining arguments are data files to send to the training algorithm. The output of NTNGramTrainer executable will be model files written into the ModelStorage directory, which is located in the AppId install directory.

```
NTNGramTrainer.exe
3
C:\Program Files\Cigital\AppId\DataStorage\Notepad
C:\Program Files\Cigital\AppId\DataStorage\Emacs
```

Figure 27 – Configuration file for training

Modification of the training configuration file can be done with another executable found in the AppId install directory. NGramConfigurationBuilder.exe (see Figure 28) was built to provide a user-friendly way to modify the NGram training configuration file.
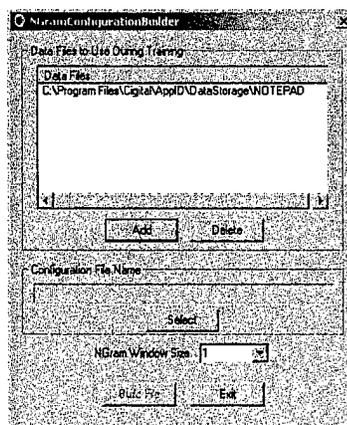

Figure 28 – NgramConfigurationBuilder Application

The user can modify the list of data files that the training algorithm will use by selecting the Add or Delete buttons. The user can chose the name of the configuration file to build by pressing the Select

button. A file selection box will be presented (see Figure 29) to select an existing file to overwrite or to enter the name of a new configuration file. Figure 30 shows a configuration file that has been selected.
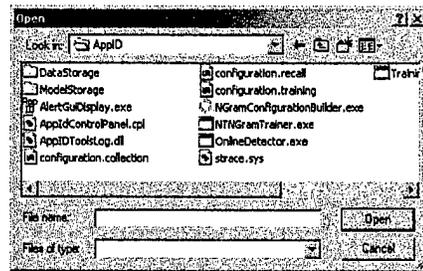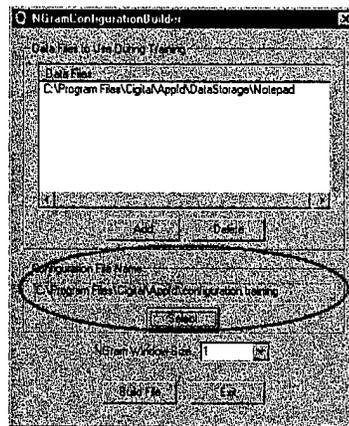

Figure 29 – Configuration File Selection Box

Figure 30 – NGramConfigurationBuilder with a configuration file given

Once the user has selected a configuration file, the window size for the ngrams can be selected (see Figure 31). The longer the window size, the more specific the recall algorithm will be. However, a longer window size will result in more potential false alarms being generated by the recall algorithm. A window size of three or less is suggested.
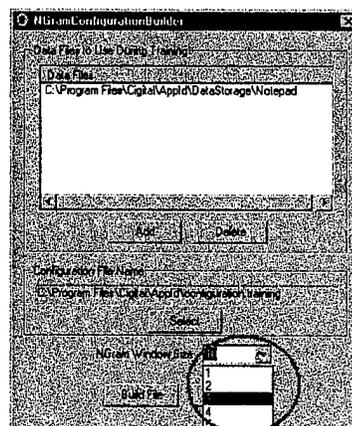

Figure 31 – Selecting the NGram Window Size

Once the window size has been selected, pressing the Build File button will build the desired configuration file. The user can then select the Exit button to exit the application.

It is possible, but not advisable, to extensively modify the configuration files that are provided with AppId. Please see Appendix A for more detailed information on the configuration files and possible components.

AlertGuiDisplay

The AlertGuiDisplay application works in conjunction with the EventLogDestination to show the user any anomalies that have been detected by AppId. The EventLogDestination component sends reports to the Windows Event Log where they will be detected by the AlertGuiDisplay application. The application can be found inside of the AppId install directory. When it is launched, it will place an icon in the system tray (see Figure 32).
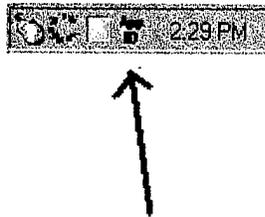

Figure 32 - AppId AlertGuiDisplay Icon in the System Tray

Selecting the AlertGuiDisplay Icon in the System tray will bring the main application window to the front (see Figure 33). The items in the list are anomalous behavior that has been detected by AppId. AppId writes the anomalous behavior into the system event log and the AlertGuiDisplay reads them out and displays the information. It will also beep when it detects new additions to alert the user.

Figure 33 – AlertGuiDisplay application main window

Appendix A

Possible Sources	
Name	NTCollector
Header	SOURCE
Previous Components	NONE
Post Components	NToID
Arguments	NONE

Table 1 - Possible Sources

Possible Components		
Name	NTtoID	NGramConfiguration
Header	COMPONENT	CONFIGURATION
Previous Components	NTCollector	NToID
Post Components	NGramConfiguration, ProcessSplitter, future algorithm configuration components,	ReportManager
Arguments	NONE	Window Size of the NGramAlgorithm

Table 2 – Possible Components

Possible Sinks		
Name	ProcessSplitter	ReportManager
HEADER	SINK	SINK
Previous Components	NToID	NGramConfiguration
Post Components	NONE	NONE
Arguments		Report Destinations (see Table 4)

Table 3 – Possible Sinks

Possible ReportDestinations		
Name	LogFileDestination	EventLogDestination
HEADER	REPORT	REPORT
Previous	Any Report	Any

Components	Destination	ReportDestination
Post Components	Any Report Destination	Any Report Destination
Arguments	Any file, or no file for output to standard error	Machine name or no argument for local machine

Table 4 – Possible Report Destinations