9-1-2006

Tools and algorithms to advance interactive intrusion analysis via Machine Learning and Information Retrieval

Javed Aslam
Dartmouth College

Sergey Bratus
Dartmouth College

Virgil Pavlu
Dartmouth College

Tools and algorithms to advance interactive intrusion analysis via Machine Learning and Information Retrieval

Javed Aslam, Sergey Bratus, Virgil Pavlu

Abstract

We consider typical tasks that arise in the intrusion analysis of log data from the perspectives of Machine Learning and Information Retrieval, and we study a number of data organization and interactive learning techniques to improve the analyst's efficiency. In doing so, we attempt to translate intrusion analysis problems into the language of the abovementioned disciplines and to offer metrics to evaluate the effect of proposed techniques. The Kerf toolkit contains prototype implementations of these techniques, as well as data transformation tools that help bridge the gap between the real world log data formats and the ML and IR data models.

We also describe the log representation approach that Kerf prototype tools are based on. In particular, we describe the connection between decision trees, automatic classification algorithms and log analysis techniques implemented in Kerf.

Chapter 1

Introduction

In this chapter we formalize the role and model of log analysis in computer intrusion investigations, and identify the common tasks that are the prime candidates for automation.

1.1 Intrusion analysis background

We consider application of Machine Learning and Information Retrieval techniques to the problem of *intrusion analysis*, the area of operational computer security that remains something of a black art. In this paper we attempt to model the actions of an intrusion analyst and suggest helpful automations. We describe the prototype implementations of these ideas in the Kerf toolkit.

Whereas some techniques used for intrusion analysis are also applied in the much more widely studied field of *intrusion detection*, there are fundamental differences between these tasks. We summarize these in Table 1.1.

	Intrusion Detection	*Intrusion analysis*
time requirement	real-time or close	post factum
context available	limited by required reaction time	all logged events
agent	fully automatic	human in the loop
definition of normal activity	pre-existing policy or statistical model	derived during analysis
definition of anomalous activity	pre-existing rules or statistical model	derived from analysis
output	labeling of observed events	hypothesized timeline of the intrusion

Table 1.1: Intrusion Analysis vs. Intrusion Detection

It should be noted that the difference in the amount of context available to an intrusion detection system for its labeling decisions and to an intrusion analyst is crucial. An IDS can only maintain a very limited amount of state for each decision because of the performance cost associated with maintaining this state information; whereas stateless systems are likely to fall prey to avoidance techniques exploiting the difference between the IDS's and the target's representation of events (see, for example, [PN98]). For post-hoc intrusion analysis such limitations are largely removed, while other factors, characteristic of IR and ML problems, come into play.

In this paper we focus on the information that can be gained from log data (which is, of course, only one possible source for an intrusion analyst, besides direct observations of intruder activity, forensic analysis

of disk and memory images from compromised machines, binary analysis of intruder software, etc.) We suggest a number of IR and ML techniques to help the analyst with labor-intensive log analysis tasks, for which we developed prototype implementations in our Kerf toolkit, and we discuss our experience with these implementations. We also outline the data and algorithmic issues specific to host and network logs that set this class of problems apart from other areas where IR and ML methods are applied and describe auxiliary tools that we developed to address them.

The Kerf toolkit consists of a centralized relational database-backed logging framework, the *SawQL* domain-specific query language designed to succinctly express time and resource correlations between records, and GUI applications that accept SawQL queries, present their results, and allow users to manipulate, classify and annotate the records, in his search for the form of presentation in which intrusion-related records are most easily found. The user's initial view of query results and subsequent adjustments to it are driven by data organization and ML algorithms. Some of these algorithms are unsupervised, others take advantage of user feedback.

The Kerf logging architecture, query language and the correlation engine that implements it are described in detail in [gro04a, gro04b]. The data organization component, *Treeview*, is intended to help the user navigate large result sets of log records (or pairs, triples, etc. of related log records) produced by the correlation engine.

For each dataset, Treeview provides an initial hierarchical presentation, whose top level serves as a compact summary, which can be further expanded in the parts that look interesting. It does so by applying an unsupervised data organization algorithm, discussed in Section 1.3.3. Essentially, the records are folded into groups, recursively, to produce a tree and the folding order is chosen to highlight statistical peculiarities of the set. This presentation should save the user time and effort understanding the structure of the result set and expose anomalous values and correlations that exist in it.

The tool also accepts user tags for the records and their groups, and helps the user to find records similar or related to those marked, via one of its supervised learning algorithms (the user has the choice of several implemented methods to find one that best matches his data). We cover these algorithms in Section 3.1.

Kerf was designed to be largely independent of the log data source, recognizing the fact that an analyst may not have control of the format of the logs he is called upon to analyze. In particular, these logs may exist as free text, not well suited for parsing (e.g., UNIX syslog). To simplify the parsing task, we provide the *PatternHelper* tool (described in 1.2.1) that attempts to automate generation of patterns to parse the values of interest out of syslog-like free text.

We applied our tool to several real and generated datasets. The annotated test data that we used for evaluating our algorithms was generated using the DARPA Cyber Panel Grand Challenge Problem tools (GCP4.1) which synthesize IDS alerts based on a scenario where a global player is under large scale strategic cyber attack. The IDS alerts are in standard IDMEF XML format and are created from templates based on real IDS reports generated by popular IDS tools which are widely used. The simulated network includes several large computing centers and many satellite centers, which have both publicly accessable platforms (web servers, email relays) and private platforms (workstations, VPN's). The alerts created by these tools are of sufficient fidelity to be used by researchers evaluating cyber situation awareness and alert fusion technologies. Two data sets were generated, one with both attack and background alerts, and one with attack-only alerts. The attack-only data set was then used to mark the attack alerts in the attack plus background data set. The resulting marked data set was stored in a MySQL database and consisted of 56,350 records representing 1520 networked computers where 14,843 of the records were attack related.

1.2 The log analysis model

In order to make use of IR and ML tools, we must first process the log data to bring it into the suitable form. In this section we discuss the challenges involved in such transformations, and describe the *PatternHelper* tool that we developed to aid in parsing of syslog and syslog-like sources.

1.2.1 Logs: ideal and reality

We would like to adopt the view of log data taken by the Machine Learning (ML) and Information Retrieval (IR) communities, namely, that data comes in neat records with well-defined fields, and the records, thought of as *feature vectors*, can be usefully ranked and retrieved based on the values of these fields. We will follow the ML and IR terminology that describes the process of data preparation as transformations of *raw data* first into *raw named features* and then into vectors of *processed features*.

Some logs are indeed produced in this form or can be easily converted into it (e.g., the Snort IDS can log directly into a relational database), but many others start out as more or less human-readable free text, and need to be parsed.

The syslog facility in UNIX uses printf-like format strings, and we estimated that over 150 different format strings are found in the syslog of a typical RedHat FC 1 workstation. Even narrowly specialized systems such as the wireless access points produce over 20 different types of syslog messages whose format varies substantially from vendor to vendor.

Regexp-based tools such as `logwatch` and `swatch` were written to automate syslog parsing, but adapting them to new formats requires the administrator to write (and debug) a regexp for each new type of record and provides no assistance in this task. Also, while an administrator may have invested significant effort into patterns for parsing his regular logs, for an intrusion analyst there is a high chance that relevant information will be contained in free text logs.

PatternHelper. Recognizing this need for automating freetext log conversion into the *feature vector* form, we developed our heuristic *PatternHelper* tool that helps the system administrator to develop patterns for parsing a large number of syslog-type text messages, in which variable fields are generally delimited by whitespace.[1] The tool uses simple Bayesian models for most of its probabilistic decisions, and in its apllication, PatternHelper

1. inputs large amounts of log records and pre-tokenizes them on whitespace,
2. separates the resulting token sequences into groups that appear likely to have been generated from the same format string,
3. for the largest group, determines the position of variable fields, based on a dictionary of values of previously observed field values,
4. for each field, suggests a name, based on the nearby tokens, and reverse dictionaries of the previously observed values,
5. presents the user with a new pattern, hypothesized as the result of the above, for editing and confirmation, and
6. based on the user input, adjusts its dictionaries and count tables, and returns to (3) in case the size of the next largest group is greater than a threshold value and otherwise returns to (1) and requests more input data.

The accuracy of guesses can be improved by extracting strings that appear to be format strings for log messages from application binaries on the target system. The field name guesses that cannot be easily made based on the apparent type of the value (such as IP addresses or URLs) use a dictionary of commonly used English nouns and prepositions.

The resulting patterns look as follows. Named fields in which whitespace is allowed are prefixed with `%%`, others with a single `%`. For most fields in this example, PatternHelper came up with the correct names

[1] For classic syslog, tokenization on whitespace would sometimes lead to errors, because whitespace in values such as filenames is not escaped or delimited by any standard convention. Consequently, it is to distinguish whitespace present in the format string to separate tokens from whitespace that is part of a field value. Statistically, however, most whitespace in syslog serves to delimit tokens.

for the field, derived from the surrounding tokens, but some user intervention was required for clarity, for example, in changing the `host` field in the second rule to `src_host`.

```
#-------   ipop3d:     --------------------------------
ipop3d:
  Login user=%%user host=[%host_ip] nmsgs=%num1/%num2
  Auth user=%%user host=%src_host [%src_ip] nmsgs=%nmsgs
...
#-------   sshd:     --------------------------------
sshd:
  Failed %auth for illegal user %%user from %src_ip port %port
  Accepted %auth for %%user from %src_ip port %port
...
```

Here the overall format of input lines is assumed to be `syslog(3)`, so a fixed common regexp pattern is used to match the timestamp, the hostname, and the service `ident` string (possibly followed by the process id or other useful information included in the `ident` tag by the application). Based on such patterns, the free text input is parsed into feature value tuples, forming records of named features. The resulting records, augmented with processed features if necessary, are inserted into a relational database.

1.2.2 The log analysis model

In the above terms, we describe the general intrusion analysis problem as follows. A certain unknown subset $\mathcal{A}$ of the set of all available records $\mathcal{R}$ consists of records relevant to the intrusion (the initial penetration and the subsequent activities of the attacker). We define the goal of the intrusion analyst as finding all the records of $\mathcal{A}$ in $\mathcal{R}$. The completeness requirement is essential for analysis, since an undetected relevant record may result in an underestimated scope of the compromise. We assume that the human analyst can recognize most attack records directly, or distinguish them from others based on context and the previously found subset of $\mathcal{A}$, *once he is presented with them*. Thus a major purpose of intrusion analysis tools is to organize the data and present it in such a way as to maximize the chance that the analyst will encounter the records from $\mathcal{A}$ in his browsing.

Our toolkit approaches this task from several different directions. Firstly, the data is initially presented in a hierarchical form so that similar records likely to have the same relationship to $\mathcal{A}$ are folded together. Thus a large class of similar records would not distract the analyst's attention from smaller, possibly anomalous classes. Secondly, the hierarchy is chosen (by an unsupervised data organization algorithm) to favor the pattern when the majority of records can be folded away into a small number of groups, leaving the anomalous "tail of the distribution" cases more visible from the outset. Thirdly, user feedback is utilized to re-rank the records based on a correspondingly adjusted measure of similarity (the supervised learning approach). Finally, the active learning prototype component presents the user with individual records, which appear important based on the user markup and previous classification.

1.3 Data organization

One of the fundamental tasks of log analysis is browsing large amounts of records or their tuples (returned by a correlating query, e.g., pairs of logged request-response events) for apparent anomalies. In practice it is usually hard to rigorously define what constitutes an anomaly in the logs, even though the system administrator may immediately recognize it as such, e.g., an unusual value of a feature (say, a source IP or payload length of a request), an unusual frequency count on a record, or an unusual dependency between values of two features (say, an unusual pattern of connections for a machine or a user). In many cases, the analyst does not have a baseline picture of "normal" system behavior and needs to establish both the "normal" behavior statistics and, at the same time, to spot and classify the outliers. In other words, the analyst's descriptions of "normality" and "suspiciousness" change based on ongoing observation.

We model this process as the construction of a decision/classification tree with records as leaves and functions of features as predicates in the intermediary nodes, determining the place of each record in the tree. The choice of these functions, i.e., of the manner in which the records are recursively grouped, is driven by the desire of the analyst to separate the "normal" records, undoubtedly in $\mathcal{R} - \mathcal{A}$, from those that merit further investigation. Thus the tree structure should separate the "normal" from "suspicious" (likely in $\mathcal{A}$) as close to the root of the tree as possible, so that heavy intermediary nodes near the top of the tree would correspond entirely to "normal" behavior (i.e. all of their leaves in $\mathcal{R}-\mathcal{A}$), or entirely to the attack (all leaves in $\mathcal{A}$), while the branching factor at the top levels of the tree remains low. In other words, the hierarchical data representation should agree with the partition of $\mathcal{R}$ into $\mathcal{A}$ and $\mathcal{R}-\mathcal{A}$ as well as possible, while keeping the number of sibling intermediary nodes small enough to permit efficient browsing.

The simplest way to hierarchically group the data is by consecutively splitting it on distinct values of a raw or processed feature at each level. In this case the analyst is interested in choosing the sequence of such splitting features that best separates the records as above. A good example of a processed feature would be a tuple of features that show an interesting correlation pattern as their values co-occur in records, such as `(user, host)` in authentication records or `(protocol, src_ip)` in IDS alerts. The fact of such correlation, as well as the outliers of the joint distribution, are usually of direct interest to the analyst. In particular, the analyst may consider a feature to be a good classifier because its distribution has a few outliers besides a small number of frequent values.

We propose a data organization technique based on the fundamental concepts of Information Theory (following their exposition in [CT91]) for choosing the sequence of splitting features.

1.3.1 Entropy and anomalies

Entropy-based measures of anomaly for feature distributions have been successfully used for intrusion detection (e.g., [LX01]). We use such measures for the task of hierarchical data organization rather than for flagging anomalous records, but similar intuition applies.

Consider the frequency of user logins to a system. Given frequency counts $f_1, f_2, \ldots, f_n$ for n such possible events, and $f = \sum_{i=1}^{n} f_i$, the total number of such events in $\mathcal{R}$, the entropy of the feature $\bar{f}$ is

$$H(\bar{f}) = \sum_{i=1}^{n} \frac{f_i}{f} \log_2 \frac{f}{f_i}.$$

The *perplexity* of $\bar{f}$, $2^{H(\bar{f})}$, can be viewed as the effective number of different values of the feature $\bar{f}$ present in the data. Thus if $\bar{f}$ in the above example is user identity, the vast majority of login events will correspond to $2^{H(\bar{f})}$ users. If we observe that $2^{H(\bar{f})} \ll n$, then we know that users fall into several groups with respect to their login frequencies. Some of these logins may be anomalous, and the analyst may want to further examine the distribution of other features across these groups to classify them.

The *mutual information* $I(\bar{f};\bar{g})$ between two features $\bar{f}$ and $\bar{g}$ is defined as

$$\begin{aligned} I(\bar{f};\bar{g}) &\stackrel{def}{=} H(\bar{f}) + H(\bar{g}) - H(\bar{f},\bar{g}) \\ &= H(\bar{f}) - H(\bar{f}|\bar{g}) \\ &= H(\bar{g}) - H(\bar{g}|\bar{f}) \end{aligned}$$

where $H(\bar{f},\bar{g})$ is the entropy of the joint distribution of these two features, i.e., of the pairs of co-occurring values observed (f_i, g_i) across $\mathcal{R}$. The values $H(\bar{f}|\bar{g})$ and $H(\bar{g}|\bar{f})$, for which the above equation can be considered as their definition, are called *conditional* entropies and represent the uncertainty that we have about $\bar{f}$ knowing $\bar{g}$ and about $\bar{g}$ knowing $\bar{f}$, respectively.

Mutual information is used to characterize the correlation between values of $\bar{f}$ and $\bar{g}$, or, more precisely, to quantify how much information observing a certain value of $\bar{f}$ gives us about the value of $\bar{g}$ in the same record. In the above example, if $\bar{g}$ is the host on the local network to which the user logs in, then a high value of $I(\bar{f};\bar{g})$ would suggest that most users strongly prefer particular machines (while a low value indicates

that users in general tend to use the machines uniformly, without a particular preference pattern). A user contradicting the common pattern may be of interest to the analyst, so grouping the login events in $\mathcal{R}$ by the value of `(user, host)` may be useful.

Thus we can see how facts about entropy and mutual information of feature distributions over $\mathcal{R}$ can serve the analyst as useful leads in his handling of log data and locating the records in $\mathcal{A}$. In the following sections we will suggest ways to build data organization tools in which the presentation of data to the user will be driven by such facts. However, before we introduce the formal framework in Section 1.3.3, we will consider a characteristic example of the current analysis practices, and we show that these practices converge to many of the same intuitions (although taking advantage of them requires a lot of trial-and-error grunt work which we aim to automate).

1.3.2 Anomaly browsing

The kind of log analysis that we refer to as *anomaly browsing* is best illustrated by the Burnett tutorial [Bur03]. In this example, the administrator is searching web server logs for clues regarding a break-in that occurred some time in the past, supposedly through a CGI script vulnerability.

The log records exist as records in a relational database table and are extracted with SQL queries. Thus the columns of the table are raw features, and a small amount of processing with SQL's functions turns them into processed features. In order to locate the traces of the attack, the administrator repeats the following sequence of steps, which we believe is characteristic of this browsing pattern:

1. Chooses a feature or a pair of features from the feature (column) set.
2. Groups all records by the distinct values of the chosen feature or composite feature.
3. Examines the resulting values according to their frequency distribution. Typically the items of interest are found either among the top few or the bottom few in the frequency order.
4. Discards the choice of feature(s) if the distribution looks uniform, or no statistical anomaly is apparent, or the values are too numerous to examine, and repeats the above steps, *or* uses the anomalous value(s) to refine the set of considered records by a `WHERE` clause or subquery, and continues the analysis on this smaller set.

Formalizing this scheme, we suggest that automatically ranking the possible choices of features and their simple combinations to favor ones with the above properties and using them to organize the data for the user will save significant amounts of time and effort often spent manually looking for the usable presentations of record sets while browsing for anomalies.

It is interesting to notice that the groupings that are investigated further are those with a small number of frequent values, and refinements result from those distributions which would be favored by our entropic ranking defined below.

1.3.3 Entropic splitting

We now formalize the intuitions from the previous section using models based on information theory metrics.

Given a record set $\mathcal{R} = \{R_1, R_2, \ldots, R_N\}$, let $\bar{f} = \{f_1, \ldots, f_n\}$ be a feature which takes on n different values with frequences $f_1, \ldots, f_n$ and let $F = \{\bar{f}_j\}$ the set of all such features, raw or processed, available for the record set $\mathcal{R}$. We will denote the value of a feature $\bar{f}$ of a record $R \in \mathcal{R}$ by $R(\bar{f})$.

We choose the first splitting feature as

$$\bar{f}_{k_1} = \underset{\{\bar{f} | \bar{f} \in F, H(\bar{f}) \neq 0\}}{\operatorname{argmin}} 2^{H(\bar{f})}$$

or

$$\bar{f}_{k_1} = \underset{\{\bar{f} | \bar{f} \in F, H(\bar{f}) \neq 0\}}{\operatorname{argmin}} \left(2^{H(\bar{f})}\right)^{\alpha} \cdot \left(\frac{distinct(\bar{f})}{N}\right)^{1-\alpha}$$

where for a feature $\bar{f}$ the entropy and the number of distinct values $distinct(\bar{f})$ are taken for the distribution of the values of $\bar{f}$ over $\mathcal{R}$, and α is a weighting constant. We exclude features with zero entropy because those features have only one value, common for all the records in $\mathcal{R}$ and thus cannot serve to distinguish between records.

The intuition behind this method is to choose the feature f that has the least *effective* number of distinct values, measured by the perplexity $2^{H(\bar{f})}$ of its distribution. This favors distributions where the weight of the distribution is spread between a few values, possibly with some outlying values as well. The second factor in the second formula adds a factor that disfavors distributions with too many distinct values even though their perplexity is low, on the grounds that creating too many sibling groups for infrequent values of a feature may be inefficient for the analyst, who should not be made to read through many screenfuls of data if it can be avoided.

We then perform the same choice recursively to obtain a hierarchical grouping of $\mathcal{R}$. The resulting intermediate nodes are labeled with the fixed distinct values of the features previously chosen, which uniquely define the path to the root of the tree, plus representations of the ranges of the remaining features over the subset of $\mathcal{R}$ that forms the leaves under that intermediary node. The label serves as a summarization of the record group; other methods of summarization can be chosen as a customization.

An alternative method consists in constructing the chain of splitting features $(\bar{f}_{k_1}, \bar{f}_{k_2}, \ldots)$ such that

$$\bar{f}_{k_i} = \underset{\{\bar{f} | \in F, H(\bar{f}) \neq 0\}}{\operatorname{argmax}} H(\bar{f} | \bar{f}_{k_{i-1}}), \quad i > 1.$$

This will order the features by their mutual information with the first splitting feature chosen as above. When this method of hierarchical grouping is used, the correlations between feature values become apparent. When the mutual information $I(\bar{f}_{k_i}; \bar{f}_{k_{i+1}})$ between two features adjacent in this ranking exceeds a specified threshold, we merge these features into one composite tuple feature, and adjust the grouping and the labels accordingly.

Treeview. We implemented the above entropic rankings in *Treeview*, our data organization tool for the Kerf toolkit. It receives the result set of a user query and produces its hierarchical representation based on the splitting algorithm outlined above, then displays it in a standard GTK tree control.

An example of such presentation can be seen in Figure 1.1, where the hierarchical grouping chosen by our data organization algorithm makes apparent the correlation between user identities and the hosts of origin for SSH login authentication records in an actual (anonymized) log fragment. For example, it becomes obvious that the bulk of successful logins (589 out of 600) were made by a single user from one and the same host. Additional grouping is performed to conserve screen space.

The initial presentation of the data set produced by Treeview is designed to help the user understand the overall structure of the result set while presenting him with as few screenfuls of data to scroll though as possible, and also to alert him of correlations between feature values. The same algorithms can be applied to any subtree, to refine the classification.

The user can build his own classification taking this tree for a starting point, by creating new nodes with his own predicates, applying different grouping algorithms to rebuild some subtrees (or the entire tree), saving and loading previous tree templates, adding new records to the existing tree (the added records will be highlighted), sort sibling nodes on precomputed and new features, export the records in a subtree to a different instance to treeview to apply a different splitting algorithm and cross-reference the results, and perform other operations. Treeview also gives the user several modes of tagging the nodes. This user markup is used by the learning modules of Treeview to adjust ranking of records or the shape of the tree.

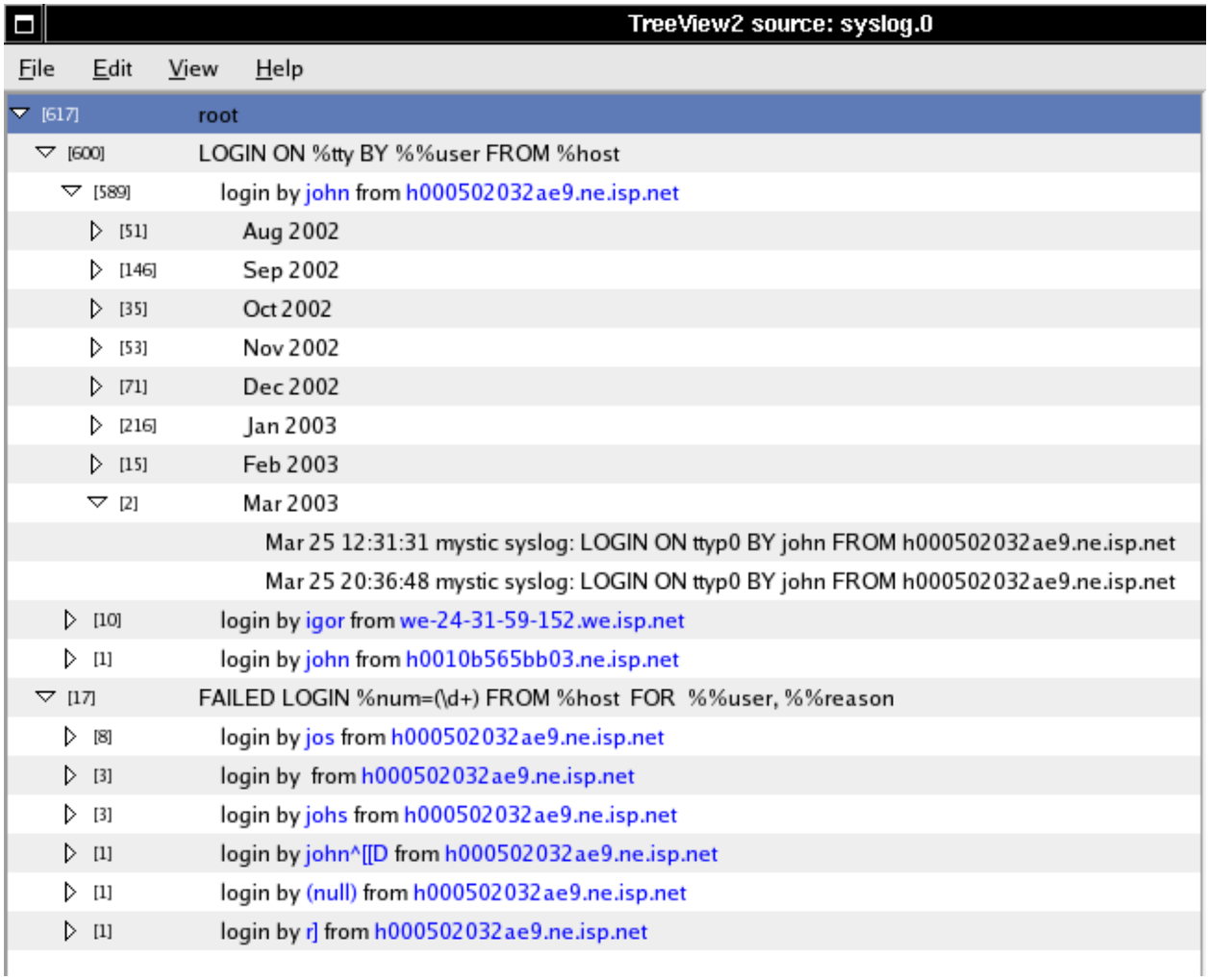

Figure 1.1: Treeview data presentation shows high correlation between user name and host of origin for SSH logins.

Chapter 2

Log as a computational object

In this chapter we describe our approach to representing logs and discuss its connections with other useful developments in computer security. We proceed to outlining our design for constructing such representations of logs.

2.1 Introduction

A number of approaches to storing, presenting and querying logs have been tried to take log analysis beyond mere manual examination of text files. In our previous technical report we gave a detailed overview of those. To summarize, existing practical approaches can be divided into several broad categories:

1. The log is considered as a stream of text records, to which regular expressions are applied, and the results are piped through a number of filters, sorted and grouped to derive a statistical summary of the information contained in the log, or to find the records that match a combination of patterns.

2. The log is considered as a relational database table, and SQL queries are run against it to extract records that fit hypotheses expressed in the relational data model, primarily as joins of tables and views. We note that the choice of this representation and SQL as a query language *de facto* determines the choice of the class from which hypotheses are drawn.

3. The log is considered as a set of records treated as in the classical Information Retrieval problem of locating records relevant to a query and ranking them according to their relevance. This approach is becoming popular with the success of search engines that follow this model for general documents (e.g., the Splunk product[1])

We propose a new model for representing logs that combines elements of the above approaches. We treat the log as a tree-like hierarchical object, in which position of an individual record is determined either by a set of rules that determine grouping and ancestry relations between records and record groups, or by statistical algorithms, based on the distributions of record field values across the entire data set or its subsets. Queries applied to this log object take into account its hierarchical structure; they can be thought of as queries on trees, with primitives borrowed from the Xpath language.

The reference to trees and tree-based languages is not incidental. A decision tree has become the standard way of formulating and representing security policies.

For example, the Netfilter/IPtables GNU/Linux firewall architecture[2] has evolved from the flat set of rules that were applied to incoming traffic in sequence, the first rule matching a packet resulting in accepting or rejecting it, to the full decision tree structure that supports user-defined branches (so-called *chains*) that

[1] http://www.splunk.com
[2] http://www.netfilter.org/

can be arranged to form trees of arbitrary depth and complexity[3]. Whereas IPtables is one of the best examples of this approach, other policy languages (e.g., various ACL languages) started following the same design principle.

The intention of such architectures is to match the often complex concepts involved in classifying network traffic to and from an organization's network; the same applies to other kinds of monitored events.

Most policies involve some kind of logging in connection with their rules, especially rejecting ones. Accordingly, this brings us to the following important observation: *The log records resulting from a decision tree structured policy, acquire a natural tree structure themselves. Conversely, a tree structure on the logs suggests a decision tree-like policy.*

In this paper, we focus on algorithms for constructing log tree objects from a flat log. Queries and transformations of trees have been well-studied outside of the log analysis community, and we refer the reader to Xpath[4] and XSLT[5].

2.2 Simple example: system call logs

Our early experiments were with system call logs. Such logs were the object of a number of studies (e.g., [LS00, LNY+00]), prompted by the IDS evaluations based on the famous Lincoln Labs datasets. We realized that a natural way of organizing, summarizing and querying syscall records was to arrange them in a tree structure in which systems calls made by a process were arranged sequentially in the order they were made, whereas system calls that created new processes (such as *exec, fork, vfork*) served as nodes where the call sequences of the corresponding child process were attached. The result was a tree of system call records arranging all records in a single structure.

Our prototype system worked with both Solaris BSM logs (the format used by the Lincoln Labs datasets), and with the GNU/Linux Snare [6] syslog traces. We had to modify Snare to support logging of all system calls that created new processes, to maintain tree connectivity. Views produced by our software combined the information provided by the UNIX tools *pstree* and *strace*.

This tree of syscalls could be filtered by the types of syscalls or by the resources (files or sockets) accessed. It provided summaries of the system's activity, and was helpful in reverse engineering and behavioral analysis of malware. We used a CGI interface to implement it, with forms to specify custom filters and multiple hyperlinks to other views and pre-written filters. The filter specification language allowed color-coding schemes for better visualization of syslog trees.

A working demo of our syscall log browsing tool can be found on the project website[7]. Various views of the syscall logs generated by the tool are presented in Figures 2.2– 2.2. The first two views show summarizations of the system call log by process, whereas others show the reconstructed system call tree, first filtered to show only the process creation related calls, and then all logged calls.

[3]More precisely, a matching rule can either accept or reject the packet, or hand it off to a new chain, that is, a new sequence of rules defined by the user; a packet may traverse many chains before hitting a rule that decides its fate.

[4]http://www.w3.org/TR/xpath

[5]http://www.w3.org/TR/xslt

[6]http://www.intersectalliance.com/projects/Snare/

[7]http://kerf.cs.dartmouth.edu/syscalls/

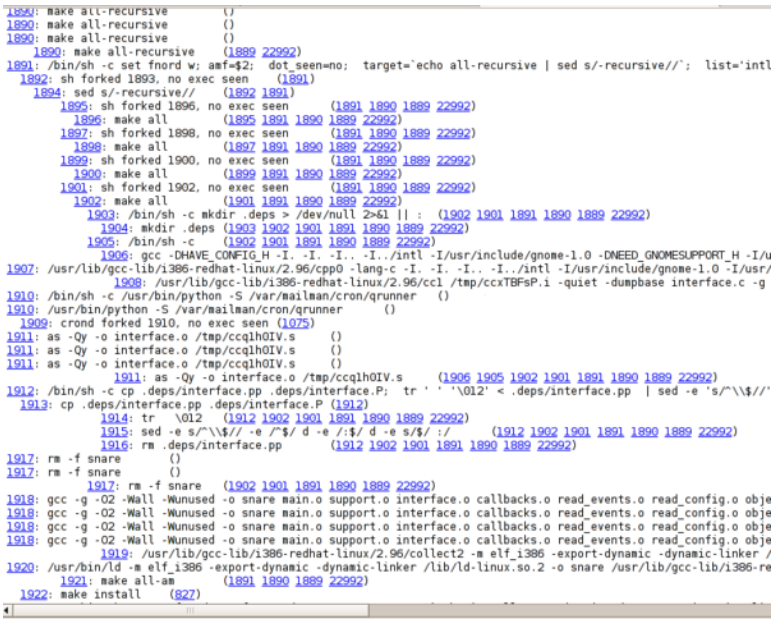

Figure 2.1: Summary process view by *exec* command

Figure 2.2: Summary system call view by process

Although we later concentrated on other types of logs, the basic principle of using tree structures and queries to organize and filter log records remained central. With system call logs, we followed a rigid ordering of records based on their temporal sequence and process ancestry. With other types of logs we moved to more flexible tree structures based on the statistical distributions and relative frequencies of unique values of record fields.

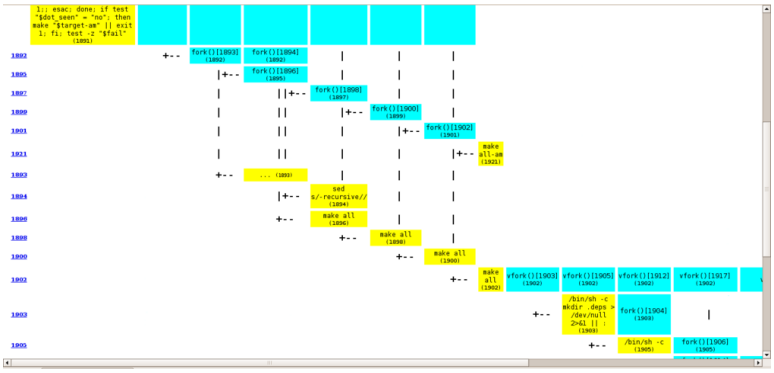

Figure 2.3: Filtered system call tree view (a)

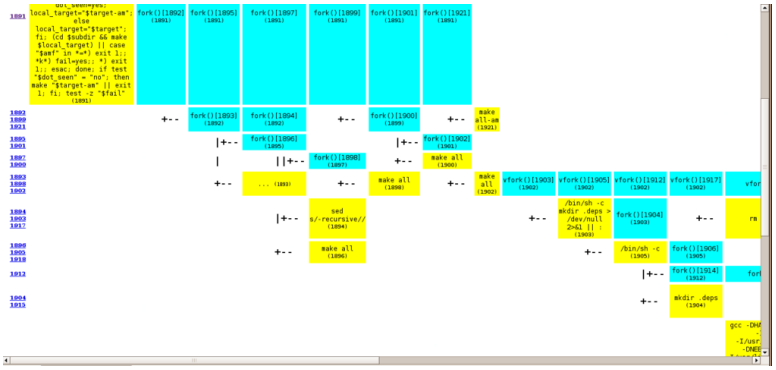

Figure 2.4: Filtered system call tree view (b)

2.3 Log trees based on unique feature values

Dealing with the UNIX *syslog(3)* logs, we observed that the temporal relationships between the records, while useful for reconstructing timelines, did not produce a useful hierarchical structure of the records. Instead, structures based on the distinct values of records fields became more important.

As a result, we designed an algorithm for constructing a tree out of a stream of logged events, as described below. We will concentrate on the rule-based systems in this chapter, and describe the adaptive algorithms

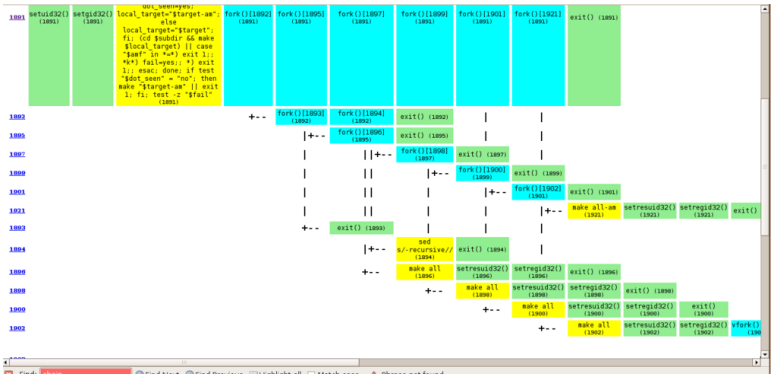

Figure 2.5: Full system call tree view

that highlight statistical anomalies in the relative frequency distributions of record fields in the next chapter.

In the following sections, we will refer to the individual rules for constructing our trees as *templates*; we chose this term because the nodes of the tree that represents the log in our viewer, *TreeView*, were generated from them in the manner similar to other template instantiation designs.

2.3.1 The tree construction model

One can imagine the algorithm for constructing the tree out of an input record set as the operation of a "coin sorter" device, in which records ("coins") pass though a number of "separator" nodes that direct them down one of the possible chutes based on the results of testing each "coin". Of course, records typically have more features to be separated on than coins do, so several layers of separators are necessary for a reasonable classification.

Taken alone, each individual separator node would sort the incoming records into a number of "bins", one per chute, based on a simple test. Their tree-like combination, where instances of other separator nodes are placed under the output chutes instead of collecting "bins", will perform a more complex sorting of the "coins" as they percolate down through these nodes, from a single entry point (the tree's root). For reasons that will soon become apparent, it is better to think of the "bins" as having no bottoms, letting the coins fall through to the next separator, but still registering their passage somehow.

As records ("coins") pass through a node (specifically, a "bin" node), it counts them and displays some form of a summary of the subset of records that ended up under it. When this mechanism is done with a set of "coins", these summaries displayed on the intermediate "bin" nodes should help the user in his searches and browsing by helping him choose the right bin to drill down to at each level of the tree.

In the TreeView display, the intermediate nodes correspond to "bins" in the above scheme. Their labels, summarizing the record subset contained in the subtrees under these nodes, correspond to the summary info displayed by the "bins". The separators themselves are not shown in the TreeView, but are fully described in the template.

The most common type of the separator corresponds to a single feature of the incoming records and has a separate output chute for each unique value of that feature. Since, practically speaking, these values are not known beforehand, we speak of chutes and bins as being "created" as needed by the passage of record "coins". This corresponds to instantiating nodes from a template to accommodate a record.

Figure 2.3.1 shows an example of a two-tiered tree in which the first (and only) tier "separator" separates records based on the destination port value, and the second tier separators break the resulting groups based on their source IP value. These second tier separators can be thought of as identical instances of a "by-source-ip" separator, cloned as needed (one for each new output "bin" of the first tier separator, i.e. for each unique value of its separating feature).

In this tree, there are only two types of separator nodes. There is only one instance of the first type, because it is at the root of the tree (the root can be thought of as the "funnel" through which the records enter). The number of *instances* of the second type depends on the dataset and is equal to the number of distinct values of source IP in it.

We now look at the template for this tree as it is used by TreeView. The full template for this tree is as follows, in XML format:

```
<rootnode name="root" next="by_dst_port"
          label="Snort portscan alerts" >
   <node name="by_dst_port"
         hashkey="%dst_port"
         label="dst_port: %dst_port     src_ip: @summary(src_ip)"
         sortkey="%num_leaves"
         next="by_src_ip"
   />
   <node name="by_src_ip"
         hashkey="%src_ip"
```

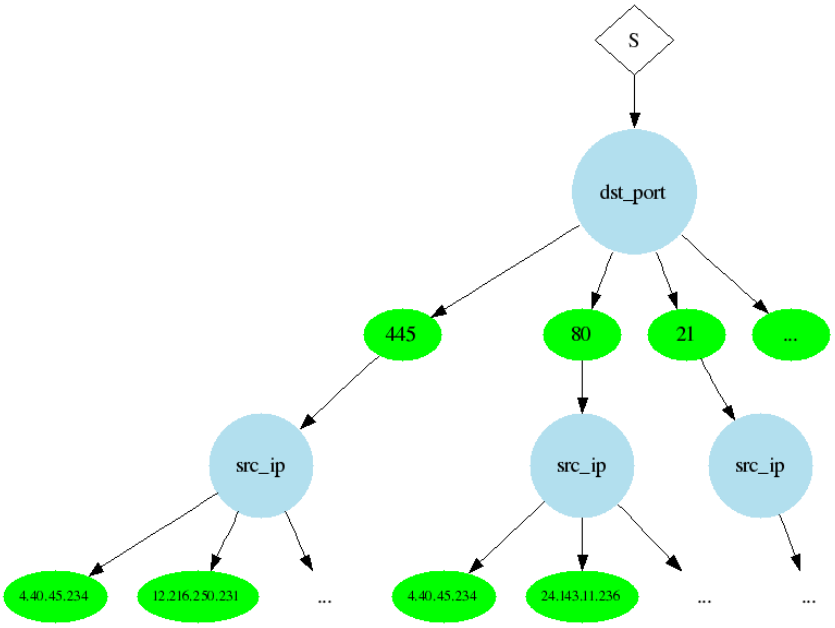

Figure 2.6: The log record tree build from a simple two-node template

```
        label="src_ip: %src_ip     dst_ip: @summary(dst_ip)"
        sortkey="%num_leaves"
        next="_line"
    />
    <node name="_line"
        label="%_line"
        terminal="1"
    />
</rootnode>
```

The two types of separator nodes are described by the two elements named *by_dst_port* and *by_src_ip*. The last element named *_line* causes a leaf of the tree to be instantiated. Notice that the top-level XML element (the so-called *document element* in XML terms) specifies the entry point (the "funnel") for the template set in its *next* attribute, and that other nodes likewise specify the template to use for the next tier of the tree.

A template's *next* attribute may contain several comma-separated template names. In that case these templates will be tried for a record in the order specified (a template may specify a *test* attribute that prevents a record from entering unless the expression contained in the *test* has a true value).

The default behavior is that templates listed in *next* are only tried until the record in accepted by one of them; however, this can be changed by specifying the attributed *pass* with a true value, in which case a copy of the record is made and continues to the next template in the list. This can be used for some forms of correlation and summarization.

Expressions in the attributes of these elements refer to the fields we expect to find in the record, in particular the self-explanatory *%dst_port*, *%src_ip*, *%dst_ip* that expand to the values of these fields, and the special *%_line* that expands to the verbatim text of the log record. These expressions can be evaluated in the context of only one ("current") record. We call this evaluation context *record context*.

Some expressions are defined only after the tree is constructed, in particular *%num_leaves* expands to the total number of leaves in the subtree under the current node. Finally, expressions like *@summary(src_ip)* refer to functions applied to the multi-set of values a feature (e.g., *src_ip*) takes on the leaves under the current

node. These are the so-called *range expressions* (the choice of the name reflects the fact that they are often used to summarize value ranges of features in subtrees). We call the context in which these expressions can be evaluated *node context*, because it assumes a node (the so-called *current node*) and a list of leaf records in the subtree under this node.

Examples of such expressions include *@max(src_port)*, the largest value of the source port under the current node, or *@H(src_ip)*, the entropy of the frequency distribution of the *src_ip* values under the current node. More information about the language of expressions permissible in different contexts of a template can be found in the TreeView manual.

Let us list the different type of "separator" nodes that can be produced from the *node* elements of the template.

1. **Hash node:** The main type of the "separator" node. This node contains an expression (the so-called *hashkey*) that is evaluated on each passing record, and places the record into the bin corresponding to the value of this expression, creating it if this value has not been seen before. The node will end up having as many children as there are unique values of the *hashkey* expression on the current record set.

   The *hashkey* attribute is required for this type of node and defines its type.

   A *hash node* can also have a *test* attribute, described below, which slightly changes its operation (only records satisfying the *test* expression are admitted into the node and go on to be separated into "bins" based on the *hashkey* value).

   The *label* attribute is required and serves to generate the labels for the "bin" nodes, the immediate children of this node.

   In the TreeView display the hash nodes themselves are hidden whenever possible, only the bin nodes are displayed, to make the tree display more compact.

   The operation of the hash node is different from other types describes below: it creates not only the "separator" but also the "bin" nodes (to which its *label* actually applies).

2. **Drop node:** Records that reach this node are dropped, incrementing a "dropped_records" counter. A *drop node* may have a *function* attribute defining a function that is called on each record before it is dropped and gathers some further statistics that gets saved in the node and displayed on its label. A *drop node* is created from a *drop* XML element *or* by explicitly providing the *type* attribute that equals *drop* on a node.

   These nodes are visible in TreeView. They have no leaves or children, and their *label* attribute is usually chosen to display some statistics about the dropped records that reached that node.

3. **Deferred node:** Leaves that reach this node immediately become its leaves without any computation, not even of a label for the leaf node created (the default label is "dummy"). This type of node is expected to contain a *command* attribute that contains the command to be run on this node after the tree construction for the input record set is finished. This command is expected to reshape the subtree under that node, so any actions during the tree construction besides the simple attachment of a record under the *deferred node* would be wasted.

   A *deferred node* is typically replaced by the results of the command (typical commands are "autosplit via *algorithm*" or "apply *saved_template_file*").

   A *deferred node* is created from a *deferred* XML element *or* by supplying the explicit *type* argument equal to *deferred*.

   Note that the *command* attribute is not limited to deferred nodes, is indeed allowed on any kind of node, and need node change the shape the tree (but may, e.g., result in marking all leaves under the current node that satisfy some condition).

4. **Leaf node:** A record that reaches a *leaf node* template becomes a leaf, labelled according to the node's *label* attribute. This attribute typically includes the full text of the record for syslog–type records, but may be an arbitrary expression for other kinds of records, e.g., packets or IDS alerts.

The standard template node named *_line* is appended to a template set by the TemplateReader parser if the input file does not contain a template with that name.

A *leaf node* template is created from a *leaf* XML element.

5. **List node:** Sometimes it is desirable to create an extra node in the tree for the purposes of summarizing a group of records satisfying a given condition. This is the intention of the *list nodes* that define no *hashkey* and do not explicitly specify any of the above types. A list node may have a *test* attribute, in which case only records satisfying the *test* expression will be allowed to enter and proceed through it; a missing *test* attribute is not an error, but means that the node is either merely decorative or serves to accumulate as leaves the records that did not pass the *test*s of its sibling nodes (see examples below).

   In the TreeView display, list nodes are visible (as intermediary nodes). The *label* is required and will be placed on that node and serves as a summary for the set of leaves in the subtree under it. A list node will also be created from a *list* XML element in the template.

   An example of a list node is the standard *_default* node, supplied for a template set by the TemplateReader if a node by that name is not explicitly defined. This template gets instantiated whenever in a series of sibling templates each one contains a *test*, and some record does not satisfy any of these tests. Instead of being quietly dropped, the record ends up as a leaf under a *_default* list node.

6. **Case node:** When placed inside a *hash node* element, a *case node* specifies a special path to take for the value of the parent *hash node*'s *hashkey* specified in the *value* attribute of the *case node*. This type of node may specify a different *label* to apply to the bin, or a different path for the record to take (in its *next* attribute, different from the parent hash node's *next* attribute), or an entirely different set of templates, valid only for the subtree under the "bin" derived from this *case*. This is done by nesting these templates in the *case* element in question.

   test attributes are not allowed on *case* nodes (the logic is simple – if a record is prevented from entering its proper bin determined by the *hashkey* value, where would it go?)

   Case nodes are specified by XML elements of type *case*.

   Case nodes result in visible TreeView "bins" among the other "bin" siblings–children of the (invisible) parent *hash node*.

The following is the list of all allowed attributes with explanations and limitations.

1. **name**: Each template node must have a name. Names starting with the underscore are reserved, and should not be used. These names are used in the *next* attributes to connect the templates and are the means of references by which multi-tiered trees of "separator" nodes can be formed.

2. **next**: Specified one or more nodes to direct the record to after the node instantiated from the current template is done with it. A comma-separated list of names is allowed here, and is expected if the first name in the list refers to a template with a *test* attribute. See also *pass* attribute.

3. **hashkey**: The expression at the heart of a *hash node* separator. To be useful, it should refer to one or mode features ("fields") of a record, and be evaluatable in the *record context* as defined above. That is, *range expressions @func_name(feature_expr)* are not allowed in hashkeys, because the hashkey should be computable in the context of a single record, whereas *range expressions* can only be evaluated in the context of a tree node after the tree construction is complete; *range expressions* apply to the set of leaves in the subtree under the node on which they are evaluated.

4. **label**: The label expressions are intended to label "bins" to which separator nodes direct records (and thus usually include the *hashkey* expression that would expand into the "bin"'s unique value of the *hashkey* expression once a bin node is instantiated), and also to show a brief summary of other feature values in the subtree under the node being labelled (and thus also usually contain *range expressions*

for the features of interest). The labels on all nodes except leaves are evaluated in the *node context*, and their evaluation is delayed until the tree for the input record set is complete.

The semantics of *label* are slightly different between the hash node templates and other template types. Namely, in a hash node the *label* is used to make the (different) labels for all the "bin" nodes, whereas in other types of nodes the *label* is used for the node instantiated from the template.

It should be noted that label on intermediate nodes must be re-evaluated when the tree changes shape as a result of a user action or command, on the entire path from the node acted upon to the root, to preserve consistency of the summaries.

5. **sortkey**: This attribute is not used while the tree is constructed, but is essential when it is displayed. The children of nodes generated from templates (in particular, of *hash* and *list* nodes) are stored in no particular order, but need a fixed order in which they can be displayed in the TreeView. This order is established by evaluating the *sortkey* expression on all the siblings, and then sorting them according to these values. Sorting is in descending order by default, ascending order is forced by setting the *sortorder* attribute to *asc*.

   The default value for *sortkey*, supplied by the TemplateReader if not present in a template, is *%num_leaves*, a special variable available in the context of a node that evaluates to the number leaves in the subtree under the current node. Another possible choice is *%num_children*, another special variable, the number of direct children of the current node.

6. **sortorder**: "asc" will cause the *sortkey* to be applied in ascending order.

7. **test**: This expression will be evaluated on a record to determine if it is allowed to enter (and instantiate, if not already instantiated) a template node. It has the same limitations as the *hashkey* attribute.

8. **terminal**: Identifies leaf templates. A "truth" value (e.g., "1") is the only meaningful one.

9. **type**: Determines the type of the node. Allowed values are *hash* (implied as long as the *hashkey* attribute is present, may be omitted), *drop*, *deferred*, *leaf* (may be omitted if *terminal="1"* is given) and *list* (implied if none of the above is given, may be omitted).

10. **value**: Allowed only for *case* templates, which are themselves meaningful only as children of a *hash node* parent. Specifies the value of the parent's *hashkey* for which the special action described by this *case* should be taken.

11. **pass**: If specified and set to a true value (e.g., "1"), this attribute specifies that the current template does not "capture" the passing records, i.e. they are duplicated and the duplicate is allowed to proceed to the next template named in the previous tier node's *next*.

    This behavior can be used in combination with drop nodes for gathering different kind of statistics with different *function*s, and then displaying them on the labels of different nodes.

12. **command**: Specifies a TreeView command that can be carried out before the tree is displayed. Nodes generated from templates with a *command* attribute are entered into a queue together with their commands, and then carried out once the tree construction from the current input set is over (in the contexts of their respective nodes). For commands that change the shape of the subtree under their current nodes, the use of *deferred* nodes is recommended to save processing time.

13. **function**: This attribute specifies a function to call immediately on the record and the node just created from the record, also passing it the value of the template's *arg* attribute as the third argument (if any). The function is passed the node and the record *by reference*, and can therefore compute and add new fields to the record or modify the existing ones, and register variables to be stored in the node for later evaluation by calling the node's *set_user_variable(key, value)* method (they will later available to the ExpressionEvaluator via the usual *%varname* syntax).

    Use with caution, because the function has access to all the node internals.

14. **postfunction**: This attribute specifies a function to call on a node once the tree is fully constructed (but before the deferred commands specified in the *command* attributes). Otherwise it is similar in operation of the *function* attribute. Note that the *function*–named function will be called for each record that reaches the node generated from the template containing the *function* attribute, whereas *postfunction* will be called only once. In typical usage, the *function* will gather information from passing records and store it in the node, whereas the *postfunction* will process and summarize it.

    Use with caution, as above.

15. **arg**: Specifies a node-specific argument for the *function* attribute. Optional (the function will get *undef* if omitted).

Using these attributes, one can create complex (possibly, Turing-complete) templates, use the saved template sets as subsets for subtrees of the constructed tree, or even write self-modifying templates.

Chapter 3

Learning Components

In this chapter we describe our applications of various Machine Learning and Information Retrieval techniques to log browsing tasks.

3.1 Introduction

Learning and data mining provide a variety of techniques for understanding, exploring, classifying and describing large amounts of data. More and more applications, particularly in security, incorporate learning/data mining algorithms. As shown in previous sections, our toolkit organizes and presents data in user-friendly fashion which makes browsing through the log data more efficient, for moderate sizes of result sets; however for large and complex logs the user (system administrator) is still confronted with a difficult task. This is where learning comes into play: the primary purpose of the learning algorithm(s) is to automatically detect and suggest records that are more interesting (i.e., likely associated with an attack) than others, therefore making the user search more efficient. A secondary purpose of the learning task is to combine the learning algorithm with an *active learning* strategy, that is a strategy to optimally select new records to expand the set of records explicitly classified by the user (the so-called *training set* in ML and IR terms), minimizing user effort in providing the marked records to derive the desired complete classification of the log data with respect to its relation to an attack.

3.1.1 Data organization

A clear distinction should be made between unsupervised learning (of which clustering is the most frequent form) and supervised learning. Clustering is used to *group similar records* to give the user a starting point for the search, when no records are yet *judged* (classified and marked) as "normal"/non-relevant or "attack"/relevant. This is the main advantage of using a clustering-like method, that it does not require any effort from the user in marking records before it can run; the disadvantage is that the record grouping produced does not always meet the user's purpose and can sometimes be misleading in the sense that the tree produced, while reflecting the general structure of the record set, is not very useful for record classification purposes with respect to a particular attack.

3.1.2 Supervised learning

Supervised learning infers classification labels for records from a *training set* of marked records that is provided by the user. This implies that the user marks a number of records *prior* to running the learning algorithm. In the classical ML and IR settings, training sets can be quite large (70% or more of the entire set), which can make the initial labelling effort difficult. In many cases training sets are based on history, like records of patients in a hospital or sales records over periods of years, and in such cases obtaining

training sets requires no extra effort. The advantage of supervised learning is that it can produce highly accurate results (1% error or less). There are many successful learning algorithms developed over last three decades, including: decision trees, neural networks, nearest neighbor, boosting, support vector machines, Bayesian networks and information diffusion. While some algorithms are very different than others in terms of performance, computation time, or the type of problem they solve, it is always the case that the larger the training set is, the better the performance.

Applying supervised learning methods to intrusion log analysis has a number of pecularities and challenges that set it apart from other applications. We discuss these in the next section.

3.1.3 Learning for intrusion analysis

For terminology we use "attack"/relevant and "normal"/non-relevant for records marked by user and "suspicious" and "non-suspicious" for inferences made by the learning algorithm. The learning task for our toolkit is for now reduced to classification of records by labelling each record with a *score* that allows our tool to rank them from the most suspicious to the least suspicious, and to present them to the user accordingly. The algorithm is given a training set (of marked records) and outputs a *real number label* for every record, indicating the degree of suspiciousness, that is, the likelihood that the record is attack-related (belonging to set $\mathcal{A}$, in terms of our proposed formalism).

The quality of the learning process is evaluated with the *average precision* measure on the list of records ranked by the suspiciousness label. Average precision (AP) [BYRN99] is the most commonly used measure in Information Retrieval for assessing the quality of a retrieved lists of documents returned in response to a give user query (consider lists returned by web search engines). This measure greatly weights retrieving relevant documents in the top ranks of a list at the expense of lower ranks; that is AP is high *if and only if* most of the relevant results are ranked near the top of the list returned. We believe it fits our learning task because this measure evaluates both the *precision* at finding records relevant to attack near the top of the returned list of results and the *recall* (the fraction of relevant records returned).

In practice, the user is likely to be able to concentrate his attention on only a small top portion of the ranked list, so our ranking, possibly combined with data organization, must work within this budget of user attention.

Very limited amounts of labelled data. Although in many applications the training set is significant in size, we cannot count on this for our application: if there are 100,000 log records, a 10% training set would require 10,000 marks by the user, which is not a feasible user effort. The biggest challenge in designing a learning algorithm for our setup is that it has to work on very small training sets (at most hundreds of marks). Grouping records, and allowing the user to mark entire groups at a time helps, but not enough to remove this challenge.

Computation time. The second challenge is computation time. Our goal is to automated an iterative and interactive process with the human user in the loop. Therefore it is important than the learning components produce results in minutes and not in days.

Self-similarity of data. Because our data sets are log records, although they may be very large, they are rather "easy" from learning point of view. For many attacks prior research has identified features that separate their traces from normal activity, and logging practices catch up to include these. Often attacks involve an abnormally repetitive pattern of accesses which are easily separated by our entropic data organization and can be classified as a whole. In particular, when an attack involves a DOS component generated by simple tools, it usually leaves many identical log records, which can be easily classified and moved out of the way.

This is why learning can be useful and fast even with very small training sets. We have implemented three algorithms: Nearest Neighbor, Information Diffusion and an IR strategy. In the sections that follow,

we discuss each of these methods in turn.

3.2 Nearest Neighbor

In its simplest form, the K-Nearest Neighbor algorithm [Mit97] first defines a distance between datapoints (often the Euclidian distance when data representation allows it), then it classifies every datapoint by taking the majority vote of the closest K training points, according to the distance. For example, K=1 means the output label of any (non-training) datapoint is simply the label of its closest marked neighbor. We have implemented two variants.

The **Discrete Nearest Neighbor** (DNN) uses the Hamming editing distance, essentially counting the number of different attributes

$$editdist(R_1, R_2) = \sum_f \delta(R_1(f), R_2(f))$$

where

$$\delta(a, b) = \begin{cases} 1 & \text{if } a \neq b \\ 0 & \text{if } a = b \end{cases}$$

and $R(f)$ denotes the value for the feature f in record R (if f=`username` then $R(f)$ is the actual username in the log record R). For classification, note that this distance can take only discrete values in the set $\{0, 1, 2, ..., |F|\}$; the label output by DNN for a record R is the majority vote of the labelled records *at minimal distance* from R. It mimics K=1, but it takes into account that there might be more than one nearest neighbor.

training set size	0.01%	0.05%	0.1%	1%
DNN AP (large set)	0.75	0.79	0.87	0.92

Table 3.1: **DNN results on a set of 110,000 records, with a component DOS attack. The running time of DNN is proportional with training set size, 80min for the largest one on 2.0Ghz 64bit machine.**

training set size	0.01%	0.05%	0.1%	1%
DNN AP (medium set)	0.73	0.83	0.88	0.97

Table 3.2: **DNN results on a set of 56,000 records, with a component DOS attack. The running time of DNN is proportional with training set size, 30min for the largest one on 2.0Ghz 64bit machine.**

3.2.1 Continuous Nearest Neighbor

Continuous Nearest Neighbor (CNN) is a variant in which the features in the distance formula are weighted

$$editdist(R_1, R_2) = \sum_f \delta(R_1(f), R_2(f)) \cdot (w_{R_1(f)} + w_{R_2(f)}),$$

where $w_{R_i(f)}$ is a weight associated with the value $R_1(f)$ of the feature $\bar{f}$ in the record $R_i \in \mathcal{R}$ (we will call this value an *attribute* of the record for short), or with the entire distribution of the feature $\bar{f}$ across $\mathcal{R}$.

Weights make some attributes or features matter more than others; for example, we might want very frequent attributes to cause small changes and infrequent attributes to cause big changes; or in authentication records for remote access to a service like `www` or `ssh`, we might want to weight changes in the features `username, dest_port` more than changes on feature `source_port`, because the latter is likely to be randomly and uniformly distributed. Weights can be automatically computed with information theoretic characteristics of feature value distribution, or can be manually set up based on prior knowledge administrator has about the system and attack. Because this weighted formula makes the distance continuous, for label computation we would not consider as in DNN the marked records *at minimal distance*; instead will take the average of the marked records at distance smaller than a constant threshold fixed by the user.

For experiments we used two log data sets. The large set has 110,000 records but only five *manually selected features* per record for a total of about 10,000 resources while the medium set has 56,000 records with 16 manually selected features per record for a total of about 5,000 resources. The tables 3.1 and 3.3 present the results of the Nearest Neighbor algorithms on the large set (110,000 records) and the tables 3.2 and 3.4 present the results of the Nearest Neighbor algorithms on the medium set (56,000 records). The discrete version of the algorithm works better because many records are simply duplicates and $K = 1$ means that all duplicates are labelled the same as their closest marked record, which is most likely one of the duplicates.

It is important to mention that a small change (even as small as 0.05) in average precision is sometimes noticeable to an experienced user by looking at retrieved results. The outstanding performance of the algorithm — notice that the best result in Table 3.2 is the AP of 0.97 — is likely due, in part, to the high degree of data homogeneity. Such a ranking list of results is almost "perfect," that is, almost all the relevant records are placed above all the non-relevant ones.

training set size	0.01%	0.05%	0.1%	1%
CNN AP (large set)	0.43	0.77	0.77	0.79

Table 3.3: **CNN results on a set of 110,000 records, with a component DOS attack. The running time of CNN is proportional with training set size, 40 min for the largest one on 2.0Ghz 64bit machine.**

training set size	0.01%	0.05%	0.1%	1%
CNN AP(medium set)	0.67	0.74	0.76	0.88

Table 3.4: **CNN results on a set of 56,000 records, with a component DOS attack. The running time of CNN is proportional with training set size, 15 min for the largest one on 2.0Ghz 64bit machine.**

The Nearest Neighbor algorithm is simple and has the advantage of providing good intuition of what it does, when it works (usually producing reasonably good results) and when it does not work; because of its simplicity, we use it as a base-line comparison. The downside is that there are certain log sets on which the algorithm fails, where the distance(s) defined above show high similarity of records (because many resources — but not all — are common), but do not separate attack records well. Also the running time of the algorithm is proportional to the size of the training set making it slow for cases with large numbers of marked records.

3.3 IR Rank retrieval (IRR)

The field of Information Retrieval (IR) is concerned primarily with extracting useful information efficiently from *unorganized* data, best known instances being web search engines like Google, Yahoo etc. Techniques developed in IR are in many cases ad-hoc in nature but highly efficient in practice.

The *vector space model* retrieval [BYRN99] uses a multidimensional representation of all documents and the user query in the *terms space*, one dimension for each term; the coordinate for every dimension (term) is often the term frequency in the document. The similarity of two documents is defined as the cosinus of the angle between the respective representations. In this paper, the algorithm implemented uses the vector model similarity; it proceeds in two stages:

Stage I First infer a smoothed probability of being related to an attack (i.e., occurring in some records in $\mathcal{A}$) on every resource; this is the same as creating the training set for IDA above except now every resource will have a associated default probability by smoothing

$$label(a) = \frac{1 + \sum_{R \ni a; R \text{ marked}} label(R)}{N + occ(a)}$$

Stage II Evaluate every record as a function of resources contained

$$label(R) = \frac{\sum_{a \in R} label(a) \cdot IDF(a)}{\left(\sum_{a \in R} label^2(a) \cdot IDF^2(a) \right)^{\frac{1}{2}}}$$

where $IDF(a)$ is a weight inverse proportional with frequency of a in the data set. This method, although its best use may require human intervention for the particular log records set and/or kind of attack shows good performance with very low running time.

training set size	0.01%	0.05%	0.1%	1%
IRR AP (large set)	0.88	0.89	0.89	0.90

Table 3.5: **IRR results on a set of 110,000 records, with a component DOS attack. A run takes approximately 30 seconds on 2.0Ghz 64bit machine.**

training set size	0.01%	0.05%	0.1%	1%
IRR AP (medium set)	0.43	0.47	0.52	0.68

Table 3.6: **IRR results on a set of 56,000 records, with a component DOS attack. A run takes approximately 30 seconds on 2.0Ghz 64bit machine.**

Results for IRR algorithm are presented in Tables 3.5 and 3.6. IRR it is by far the fastest, running in seconds for both our experimental sets. The results for the large set (Table 3.5) are very good, mainly because the records have only five features highly descriptive for attack classification. For the medium set IRR did not perform very well because there were many features (16), but also because our choice of smoothing and normalization may not have been the best.

3.4 Active Learning

Besides learning from a training set and computing suspiciousness labels for all records, learning techniques can be applied to the task of selecting new records for manual classification and labelling by the user in such a way that the impact of the user's decisions is maximized. In other words, we would like to find such records that their labels will contain the most information about the labelling of the remanining set, and explicitly ask the user to label them.

The new record labelled is going to be included in the training set; the purpose here is not necessarily to select a relevant (attack) record but rather to select a record that is useful to learn from. In an online fashion, (selecting, judging, expanding the training set) is an *episode* of active learning. Using the new training set, the learning algorithm is being rerun, then a new active learning episode takes place and so on until there are enough training records.

From an information theoretic point of view, actively selected training data means choosing the maximally informative points while randomly chosen training points are only randomly informative. It has been convincingly shown ([SOS92, Ang87, CGJ95, HKS92, ZLG03]) that training sets built with active learning strategies are very effective and that they can be much smaller than, for example, training sets obtained by randomly choosing datapoints thus requiring less user effort for bootstrapping the classification.

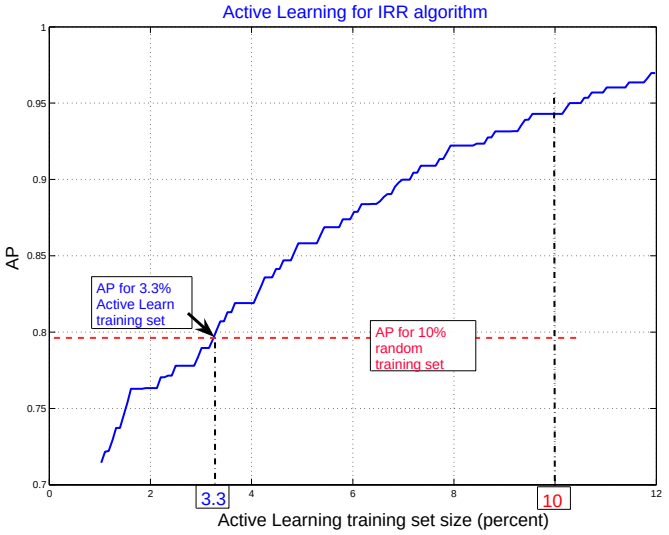

Figure 3.1: **Performance on Active Learning built training sets**

In designing an active learning strategy two factors are particularly important: first, how to choose a new record each episode and how much computation time that requires; second when training set includes the new record just marked by the user, it is desirable to find a shortcut in *updating* the learning computation instead of re-running the learning algorithm from scratch. We implemented active learning for Nearest Neighbor and IRR learning components. In both cases the record selected for labelling each episodes maximizes (in expectation) the number of records that would have their computed labels changed if the new record is included in the training set. Given the simplicity of computation formulas for both Nearest Neighbor and IRR algorithms, it is not hard to get the learning updated after an active learning episode; that only requires updates on counts on every resource marks.

In Figure 3.1 we present IRR performance on training sets built with Active Learning strategy, on a "toy" set of 1,300 records. There are two ways to interpret the advantage of active learning built training sets (blue solid line) vs the random training set (red dash line): first, the performance of a randomly chosen training set of size 10% (130 records) is matched by an actively-built training set of only 3.3% (43 records), which means same performance is achieved with an user labelling effort reduced by 66%; second, if we are to use active learning but to consider same size training set, 10%, we get a boost in performance from $AP = 0.79$ to $AP = 0.94$.

3.5 Resource graph and Information Diffusion

Accurate learning methods like Support Vector Machines or Information Diffusion often require definition of a similarity metric between datapoints, sometimes called a *kernel*; however computation for these methods is intensive because the matrices involved have N^2 elements, where N is the size of the data set. In our setup N (the number of records) can be as large as several hundred thousand therefore kernel methods cannot be applied directly. Instead, we are going to "move" the learning task from the *record space*, where N is very large to the *resource space*, where the number of resources is the sum of the number of usernames, the number of IPs, the number of ports etc., total on the order of hundreds.

We formalize this transformation of the data by introducing the *resource graph* (Figure 3.2), whose vertices are resources referred to by feature values (attributes). Two resources are connected by an edge when they co-occur in a record. Each edge thus corresponds to a subset of the set of records consisting of

records where such co-occurrence takes place, and can be labelled with a label derived from this set (e.g., with the size of the subset, or a function of timestamps). Typically resources are the feature values (or data attributes) like usernames `alice,bob` etc, ip addresses `129.170.39.103` or ports `25,80` etc.; resources can also be pairs of attributes `(alice,80)`. Even with pairs included, the total number of resources is on the order of thousands which is very small in comparison with the number of records. The resource graph is particularly important beyond the learning task because it provides a simple and complete way of modeling the log data.

3.6 Information Diffusion

The **Information Diffusion** algorithm (IDA) is a recent learning development by Zhu et al. ([ZGL03]) and it closely follows the law of *harmonic energy minimization* which in nature is realized by heat diffusion (Figure 3.4). Given this analogy, IDA propagates the information from *sources* (training examples) to the rest of the datapoints.

It uses W as similarities matrix (w_{ij}=similarity between datapoints i and j) to compute a labelling function f that minimizes the energy function

$$E(f) = \frac{1}{2}\sum_{i,j} w_{ij}(f(i) - f(j))^2$$

It is known that f has many nice proprieties including being unique, harmonic and having a desired smoothness on a graph of datapoints.

We implement IDA having resources as datapoints.The algorithm works as follows:

• given :

$W = (w_{ij})$= similarities matrix, w_{ij}=similarity between datapoints i and j

$d_i = \sum_j w_{ij}$ degrees of nodes in the graph ; D =diag(d_i)

$\Delta = D - W$ is the *combinatorial Laplacian*

• compute f = labelling function, minimizes the energy function

$$E(f) = \frac{1}{2}\sum_{i,j} w_{ij}(f(i) - f(j))^2$$

• this means $f = Pf$, where $P = D^{-1}W$ which implies f is *harmonic* ,i.e. it satisfies $\Delta f = 0$ on unlabelled set of points U. f is unique and is either a constant or satisfies $0 < f(j) < 1 \forall j$ (Boyle & Snell,1984); f harmonic means the desired smoothness on the graph

$$f(j) = \frac{1}{d_j}\sum_{i \sim j} w_{ij} f(i)$$

• exact computation: if we split W,f, P over labelled and unlabelled points, assuming labelled points get the upper-left corner, $W = \begin{bmatrix} W_{ll} & W_{lu} \\ W_{ul} & W_{uu} \end{bmatrix}$ and $f = \begin{bmatrix} f_l \\ f_u \end{bmatrix}$ then

$$f_u = (D_{uu} - W_{uu})^{-1} W_{ul} f_l = (I - P_{uu})^{-1} P_{ul} f_l$$

Using the resources as datapoints, the output of IDA is going to be a suspiciousness label for every resource which later will be combined to obtain suspiciousness labels for records. In order to apply IDA for resources we need to set up several things:

1. A similarity measure for resources. This is done using $occ(a)$ = number of records in which resource a occur and $occ(a, b)$ = number of records in which resources a and b co-occur:

$$sim(a, b) = \frac{occ(a, b) \cdot (w_a + w_b)}{occ(a)w_a + occ(b)w_b}$$

where w_a is a weight associated with resource a, often inverse proportional with the frequency of a, also called *inverse document frequency* (IDF) value.

2. A resources training set. That is, we have to label a subset of resources with suspiciousness values before running the IDA; these labels are computed from the training set of records as follows:

$$label(a) = \frac{\sum_{R \ni a; R \text{ marked}} label(R)}{occ(a)}$$

which gives 1 if and only if all labelled records containing a are marked 1 and 0 if and only if all labelled records containing a are marked 0.

3. After running the IDA we need to compute for records suspicious labels from resource labels. It can be done as a simple sum :

$$label(R) = \sum_{a \in R} label(a)$$

Tables 3.7 and 3.8 show results using the IDA on the two log sets. Even though we are using IDA in the resource space, which is considerable smaller, the algorithm can take time to run and, more important significant memory space (the largest experiment described used took 1.6GB of RAM). It has however the advantages that the results are likely accurate and that it can handle data complexity well. So if time and resources allows it, IDA is desirable.

training set size	0.01%	0.05%	0.1%	1%
IDA AP(large set)	0.80	0.82	0.84	0.90

Table 3.7: **IDA results on a set of 110,000 records, with a component DOS attack. A run takes approximately 12 minutes on 2.0Ghz 64bit machine.**

training set size	0.01%	0.05%	0.1%	1%
IDA AP (medium set)	0.67	0.82	0.85	0.92

Table 3.8: **IDA results on a set of 56,000 records, with a component DOS attack. A run takes approximately 5 minutes on 2.0Ghz 64bit machine.**

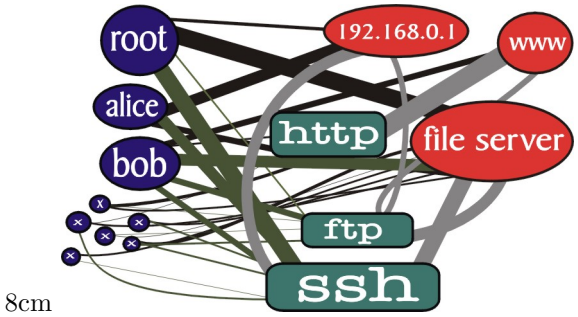

Figure 3.2: **A resource graph**

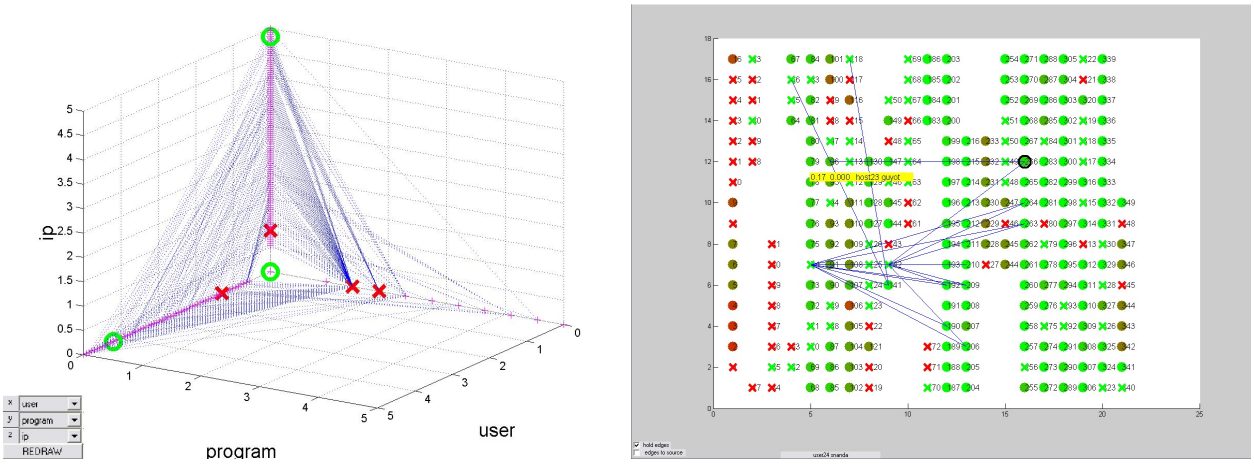

Figure 3.3: **Resource visualization. Resources "X" have computed marks from the user record marks. Red is associated with "attack" and green with "normal".**

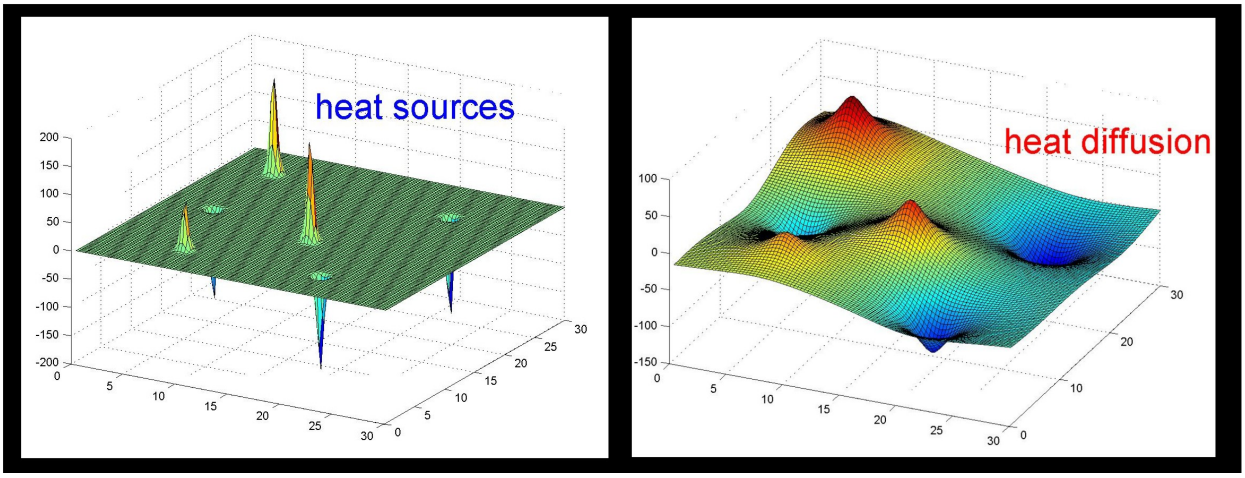

Figure 3.4: **Diffusion of heat given hot sources (up) and cold sources (down)**

Chapter 4

Related work

Several free software and commercial tools offer some form of automation for analysis of system logs from multiple sources. Tina Bird compiled an excellent survey of these, together with a collection of links to remote logging tutorials and system-specific information [Bir02]. Practical issues of log processing and correlation have also been discussed in many professional publications (e.g., [All02, Chu02, Rom00]). We draw on this and other work for the definition of applied intrusion analysis tasks that we aim to automate.

A great corpus of work has been devoted to application of ML techniques in the field of Intrusion Detection. We will not attempt to survey this work here, because most of it is based on different premises about available data, context and time limitations, as we have outlined in the Introduction. Discussions of IR and ML techniques to aid specifically in Intrusion Analysis tasks can be found in [ES02]. Various applications of visualization techniques to analysis of log data are surveyed in [Mar01]. A brief overview of available data visualization tools can be found in [She02].

The MINDS system [EEL+04, EEL+03] implemented grouping of similar IDS alerts under a single table row in its display tables, based on an association mining ML technique. MINDS data presentation is specifically tuned to IDS alert messages, while Kerf aims to provide a general technique, usable on a wide variety of log sources. Also, Kerf's records grouping is nested and hierarchical, while data views of MINDS are table-based.

4.1 Acknowledgements

We would like to thank George Bakos and Bill Stearns for many useful discussions regarding the practicalities of log analysis, Greg Conti for discussions of log visualization and connecting us with security researchers at Georgia Tech. We would also like to thank everyone who provided us with log data samples, and shared their observations on the issues encountered in analyzing logs.

This research program is a part of the Institute for Security Technology Studies, supported under Award number 2000-DT-CX-K001 from the U.S. Department of Homeland Security, Science and Technology Directorate. Points of view in this document are those of the authors and do not necessarily represent the official position of the U.S. Department of Homeland Security or the Science and Technology Directorate.

Bibliography

[All02] Jared Allison. Automated log processing. *;login:*, 27(6):17–20, 2002.

[Ang87] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[Bir02] Tina Bird. Log analysis resources. www.counterpane.com/log-analysis.html, 2002. .

[Bur03] Mark Burnett. Forensic log parsing with microsoft's logparser. http://www.securityfocus.com/infocus/1712, July 2003. .

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, 1999.

[CGJ95] D. Cohn, Z. Ghahramani, and M. Jordan. Active learning with statistical models, 1995. .

[Chu02] Anton Chuvakin. Advanced log processing. http://online.securityfocus.com/infocus/1613, August 2002. .

[CT91] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. John Wiley & Sons, Inc., 1991.

[EEL+03] L. Ertoz, E. Eilertson, A. Lazarevic, P. Tan, P. Dokas, J. Srivastava, and V. Kumar. Detection and summarization of novel network attacks using data mining. Technical Report, 2003. .

[EEL+04] L. Ertoz, E. Eilertson, A. Lazarevic, P. Tan, P. Dokas, J. Srivastava, and V. Kumar. The minds - minnesota intrusion detection system. In *Next Generation Data Mining*. MIT Press, 2004.

[ES02] Robert F. Erbacher and Karl Sobylak. Improving intrusion analysis effectiveness. Technical report, University at Albany - SUNY, 2002. .

[gro04a] The Kerf group. The Kerf toolkit for intrusion analysis. *IEEE Security and Privacy*, 2(6):42–52, November/December 2004. .

[gro04b] The Kerf group. The Kerf toolkit for intrusion analysis. Technical Report TR2004-493, March 2004. .

[HKS92] David Haussler, Michael Kearns, and Robert Schapire. Bounds on the sample complexity of Bayesian learning using information theory and the VC dimension. Technical Report UCSC-CRL-91-44, 1992. .

[LNY+00] Wenke Lee, Rahul Nimbalkar, Kam Yee, Sunil Patil, Pragnesh Desai, Thuan Tran, , and Sal Stolfo. A data mining and CIDF based approach for detecting novel and distributed intrusions. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection (RAID 2000)*, volume 1907 of *Lecture Notes in Computer Science*, pages 49–65. Springer-Verlag, Toulouse, France, October 2000. .

[LS00] Wenke Lee and Salvatore J. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Transactions on Information and System Security*, 3(4):255, 2000. .

[LX01] Wenke Lee and Dong Xiang. Information-theoretic measures for anomaly detection. In *Proc. of the 2001 IEEE Symposium on Security and Privacy*, pages 130–143, May 2001. .

[Mar01] David J. Marchette. *Computer Intrusion Detection and Network Monitoring: A statistical viewpoint*. Springer, 2001.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Secure Networks, Inc., January 1998. .

[Rom00] Steve Romig. Correlating log file entries. *;login:*, pages 38–44, November 2000. .

[She02] Brian K. Sheffler. The design and theory of data visualization tools and techniques, March 2002. .

[SOS92] H. S. Seung, Manfred Opper, and Haim Sompolinsky. Query by committee. In *Computational Learning Theory*, pages 287–294, 1992. .

[ZGL03] X. Zhu, Z. Ghahramani, and J. Lafferty. Semi-supervised learning using Gaussian fields and harmonic functions, 2003. .

[ZLG03] X. Zhu, J. Lafferty, and Z. Ghahramani. Combining active learning and semi-supervised learning using gaussian fields and harmonic functions, 2003. .