A Data Mining Framework for Constructing Features and Models for Intrusion Detection Systems

Wenke Lee

Submitted in partial fulfillment of the

requirements for the degree

of Doctor of Philosophy

in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

1999

ABSTRACT

A Data Mining Framework for Constructing Features and Models for Intrusion Detection Systems

Wenke Lee

Intrusion detection is an essential component of critical infrastructure protection mechanisms. The traditional pure "knowledge engineering" process of building Intrusion Detection Systems (IDSs) is very slow, expensive, and error-prone. Current IDSs thus have limited extensibility in the face of changed or upgraded network configurations, and poor adaptability in the face of new attack methods.

This thesis describes a novel framework, MADAM ID, for Mining Audit Data for Automated Models for Intrusion Detection. Classification rules are inductively learned from audit records and used as intrusion detection models. A critical requirement for the rules to be effective detection models is that an appropriate set of features need to be first constructed and included in the audit records. A key contribution of the thesis is thus in automatic "feature construction". Using MADAM ID, raw audit data is first preprocessed into records with a set of "intrinsic" (i.e., general purposes) features. Data mining algorithms are then applied to compute the frequent activity patterns from the records, which are automatically analyzed to generate an additional set of features for intrusion detection purposes.

We introduce several extensions, namely, *axis* attribute(s), *reference* attribute(s), *level-wise* approximate mining, and mining with *relative* support, to the basic association rules and frequent episodes algorithms. The extended algorithms use the characteristics of audit data to direct the efficient computation of "relevant" patterns. We develop an encoding algorithm so that frequent patterns can be easily visualized, analyzed, and compared. We devise an algorithm that automatically constructs temporal and statistical features according to the semantics of the patterns.

The effectiveness and advantages of our algorithms have been objectively evaluated through the 1998 DARPA Intrusion Detection Evaluation program.

Contents

List of Figures

List of Tables

Acknowledgments

This thesis would not have been possible without the help of many people. My terrific advisor Sal Stolfo has taught me to be a good and effective researcher. Sal was very patient in helping me to develop my thesis topic, and is always a big cheerleader for my progress. I would like to thank Gail Kaiser for admitting me to Columbia, providing wonderful guidance and support for my first two years here, and serving on my dissertation committee. Many thanks to Alex Tuzhilin, Henning Schulzrinne, and Danilo Florissi, for serving on my committee and providing helpful suggestions. I would also like to thank Shree Nayar for his help and encouragement throughout my Ph.D. study.

All members of our project team, Kui Mok, Chris Park, Matt Miller, Kahil Jallad, Phil Chan, Dave Fan, Andreas Prodromidis, Jeff Sherwin, Shelley Tselepis, and Grace Zhang, are great sources for ideas and fun. I thank them for making my graduate student life productive and enjoyable.

I would like to thank Naser Barghouti for being my mentor and making my summer internship at AT&T Labs Research a wonderful experience, and Jakka Sairamesh for helping me in my summer internship at IBM T. J. Watson Research Center.

I am deeply in debt to the love and support of my family. First and foremost, this thesis is dedicated to the memory of my father. He is always a great role model for me because of his devotion and excellence in scientific research, and his love to our family. His spirit will be with me forever. I also dedicate this thesis to my mother, who always encourages me to make the best effort to realize my dreams. I thank my brother for being so supportive and caring to me.

I have been blessed for having the wonderful love and support of my fiancée. She is my greatest source of comfort and encouragement, and has endured as much anxieties as I have for this thesis. I look forward to the great life ahead of us.

To the Memory of My Father

and

To My Mother

Chapter 1

Introduction

As network-based computer systems play increasingly vital roles in modern society, they have become the targets of our enemies and criminals. Therefore, we need to find the best ways possible to protect our systems.

The security of a computer system is compromised when an intrusion takes place. An intrusion can thus be defined as "any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource" [Heady *et al.*, 1990]. Intrusion prevention techniques, such as user authentication (e.g., using passwords or biometrics), avoiding programming errors, and information protection (e.g., encryption) have been used to protect computer systems as a first line of defense. Intrusion prevention alone is not sufficient because as systems become ever more complex, there are always exploitable weakness in the systems due to design and programming errors, or various "socially engineered" penetration techniques. For example, after it was first reported many years ago, exploitable "buffer overflow" still exists in some recent system softwares due to programming errors. The policies that balance convenience versus strict control of a system and information access also make it impossible for an operational system to be completely secure. Intrusion detection is therefore needed as another wall to protect computer systems.

The primary assumptions of intrusion detection are: user and program activities are observable, for example, via system auditing mechanisms; and more importantly, normal and intrusion activities have distinct behavior. Intrusion detection therefore includes these essential elements:

- Resources to be protected in a target system, for example, network services, user accounts, system kernels, etc.
- Models that characterize the "normal" or "legitimate" behavior of the activities involving these resources;
- Techniques that compare the observed activities with the established models. The activities that are not "normal" are flagged as "intrusive".

Researchers have developed two general categories of intrusion detection techniques. In *misuse detection*, well-known attacks or weak spots of the system are encoded into patterns, which are then used to match evidence from run-time activities to identify intrusions. In *anomaly detection*, normal behavior of user and system activities are first summarized into normal profiles, which are then used as yardsticks so that run-time activities that result in significant deviation are flagged as probable intrusions. Many current intrusion detection systems (IDSs) have included both misuse and anomaly detection components, and are generally complex and monolithic.

While *effectiveness* (i.e., accuracy) is the essential requirement of an IDS, its *extensibility* and *adaptability* are also critical in today's network computing environment. There are multiple security "weak links" (i.e., "penetration points") for intrusions to take place in a network system. For example, at the network level, carefully crafted "malicious" IP packets can crash a victim host; at the host level, vulnerabilities in system software can be exploited to yield an illegal root shell. Since activities at different penetration points are normally recorded in different audit data sources, an IDS often needs to be extended to incorporate additional modules that specialize on certain components (e.g., hosts, subnets, etc.) of the network system. The large traffic volume in security related mailing lists and Web sites suggest that new system security holes and intrusion methods are continuously being discovered. Therefore it is imperative that IDSs be adapted to new attack methods frequently and in a timely manner.

Currently, building an effective IDS is an enormous knowledge engineering task. System builders rely on their intuition and experience to select the statistical measures for anomaly detection [Lunt, 1993]. Experts first analyze and categorize attack scenarios and system vulnerabilities, and hand-code the corresponding rules and patterns for misuse detection. Because of the manual and ad hoc nature of the development process, current IDSs have limited extensibility and adaptability.

1.1 Problem Statement and Our Approach

This thesis research studies the problem of:

How to automatically and systematically build adaptable and extensible intrusion detection systems

We seek an *automatic* approach so that we can eliminate the manual and ad hoc elements from the development process of IDSs. We also seek a *system-*

atic approach so that the same set of development tools can be readily applied to different audit data sources. We evaluate the utility of our development approach using not only the *effectiveness* of the resultant IDSs, but also their *adaptability* and *extensibility*.

We take a data-centric point of view and consider intrusion detection as a data analysis process. Anomaly detection is about identifying the abnormal usage patterns from the audit data, whereas misuse detection is about encoding and matching the intrusion patterns using the audit data. The central theme of our approach is to apply data mining programs to the extensively gathered audit data to compute models that accurately capture the *actual behavior* (i.e., patterns) of intrusions and normal activities. This automatic approach eliminates the need to manually analyze and encode intrusion patterns, as well as the guesswork in selecting statistical measures for normal usage profiles. It is a systematic approach because the same set of data mining tools can be applied to any appropriately preprocessed audit data. More importantly, data mining tools can be applied to multiple streams of *evidence*, each from a detection module that specializes on a specific type(s) of intrusion or a specific component (e.g., a mission-critical host) of the network system, to learn the combined detection model that considers all the available evidence. The resultant hierarchical combined detection models are easily adaptable and extensible.

We have developed a framework, MADAM ID, for Mining Audit Data for Automated Models for Intrusion Detection. MADAM ID consists of classification and meta-classification [Chan and Stolfo, 1993] programs, association rules [Agrawal *et al.*, 1993] and frequent episodes [Mannila *et al.*, 1995] programs, and a feature construction system. The end products are concise and intuitive rules that can detect intrusions, and can be easily inspected and edited by security experts when needed.

Using MADAM ID, the inductively learned classification rules replace the manually encoded intrusion patterns. System features in the detection models and statistical measures in normal profiles are automatically constructed using the frequent patterns, i.e., association rules and frequent episodes, computed from the audit data. Meta-learning is used to learn the correlation of intrusion evidence from multiple detection models, and produce combined detection models.

1.2 Thesis Contributions

This thesis research contributes to both the data mining and intrusion detection fields.

- **Extentions to the Association Rules and Frequent Episodes Algorithms** We study how to incorporate domain knowledge into the basic association rules and frequent episodes algorithms so that "relevant" patterns can be computed efficiently. We utilize schema-level information about audit data as various forms of "constraints" in mining frequent patterns.
  - ***Axis* attributes** There are some attributes that are "essential" in describing (or discriminating) the data items. We argue that "relevant" association rules should describe patterns related to the essential attributes. Depending on the objective of the data mining task, we can designate one (or several) essential attribute(s) as the *axis* attribute(s)

[Lee *et al.*, 1998], which is used as a form of item constraint in association rule mining. During candidate generation, an item set must contain value(s) of the axis attribute(s). When axis attributes are used, the frequent episodes algorithm first finds the frequent associations about the axis attributes, and then computes the frequent sequential patterns from these associations. Thus, the associations among attributes and the sequential patterns among the records are combined into a single rule.

– ***Reference*** **attributes** Some essential attributes can be the *references* of other attributes. These reference attributes normally carry information about some "subject", and other attributes describe the "actions" that refer to the same "subject". When mining these frequent "same-subject" patterns, we need to filter out the unwanted "across-subject" patterns. We use *reference* attribute(s) [Lee *et al.*, 1999a] in the frequent episodes algorithm to ensure that, within each episode's minimal occurrences, the records covered by its constituent item sets have the same reference attribute value.

– ***Level-wise*** **approximate mining** We introduce an iterative *level-wise* approximate mining procedure to uncover the low frequency but important sequential patterns [Lee *et al.*, 1998]. The idea is that once episodes about a high frequency axis attribute value are produced in an early round of mining, which has high minimum *support* requirement, the value is considered "old" and can be part of an episode rule in the later rounds only if the episode contains "new" axis attribute values.

  This scheme prevents the generation of an explosive number of episodes about the high frequency values, while keeping the sequential "context" of low frequency values.

  - **Mining with *relative frequency (support)*** We also introduce a more elaborate and flexible procedure that uses *relative support* to uncover sequential patterns. Instead of the number of records in the database, the number of occurrences of each unique attribute value in the database can be used as the reference when calculating the support value of a pattern. This procedure essentially computes the top percent frequent patterns relative to each unique attribute value. Each attribute can have a different *relative support*.

- **Techniques for Pattern Visualization and Comparisons** We study the problem of presentation, analysis and comparison of frequent patterns. We develop an encoding scheme to convert each frequent pattern to a number so that we can easily visualize (and thus understand), and efficiently compare the patterns. We first define a "similarity" measure based on the "order of importance" of the attributes so that patterns that are structurally and syntactically more "similar" are mapped to closer numbers. We use a simple method to calculate the *difference* between two patterns based on their encodings. We develop a procedure that can encode a set of patterns from normal activities and another set of patterns from intrusive activities, compute the differences between the two, and output the most likely "intrusion-only" patterns.

- **Techniques for Automatic Feature Construction from Patterns** We study the problem of how to automatically construct temporal and statistical features from a frequent pattern. We develop an algorithm that parses a frequent pattern and uses *count*, *percent*, *average* operators on the occurrences of attribute values to generate features [Lee *et al.*, 1999b]. Given an "intrusion-only" pattern, this algorithm can capture both the "anatomy" and "invariant" behavior of the attack, and produce predictive features that are used to build classification models.

- **Techniques for Efficient Real-time Execution of Detection Models** We study the problem of how to efficiently execute the off-line learned detection models in a real-time environment. We seek a speed-up over linear-order rule checking (as done in an off-line setting). We define a time-based cost model for the features and rules. We develop techniques that search for low-cost "necessary" conditions for the rules, and use these conditions to filter out, in real-time, the majority of features and rules that need to be computed [Lee *et al.*, 1999b]. The results are preliminary.

- **Objective Evaluation** We participated in the 1998 DARPA Intrusion Detection Evaluation program. We applied MADAM ID to the vary large amount of raw audit data and produced automatically learned models. The objectively evaluated results showed that our intrusion detection models have the best performance when compared with other participating systems, which are all knowledge engineered. To the intrusion detection field, we showed that MADAM ID can automate many of the knowledge- and labor- intensive steps in building intrusion detection models, and demonstrated its advantages and great potentials. To the data mining field, we showed that data mining techniques can be extended and developed to be indeed very useful to an important, challenging, and traditionally knowledge engineering application area.

1.3 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 examines in more detail the challenges of building an intrusion detection system, and reviews representative research efforts. Chapter 3 gives a brief overview of the data mining process, discusses the rationale for applying data mining techniques for intrusion detection, and reviews relevant algorithms. Chapter 4 describes our experiments in building classification models for *sendmail* and network traffic, and motivates the need for automatic feature construction. Chapter 5 describes the algorithms for mining frequent patterns from audit data, with an emphasis on the extentions to the basic association rules and frequent episodes algorithms. Chapter 6 discusses how to analyze and compare frequent patterns, and automatically construct features from patterns. Chapter 7 describes experiments in using MADAM ID to build various detection models using the DARPA data. Chapter 8 describes our approach of automatically translating off-line learned models into real-time intrusion detection modules, discusses solutions to the efficiency problem, and report our initial results on experiments with NFR, a real-time network IDS platform. Chapter 9 summarizes the thesis and outlines ideas for future work.

Chapter 2

Intrusion Detection

It is well known that computer and network systems all have design flaws that can lead to security hazards [Bellovin, 1989; Grampp and Morris, 1984]. The large traffic volume in security related newsgroups and mailing lists such as the *CERT* advisories, *bugtraq*, *alt*.2600, and *Phrack* suggest that determined intruders can exploit the security flaws and break into computer systems, and it is expensive and nearly impossible to fix all the design and programming errors. Considering the fact that prevention-based approaches can not be completely relied upon, intrusion detection is needed as a last line of defense. The main benefits of an intrusion detection system include:

- Real-time reporting of break-ins so that security staff can take the appropriate actions, e.g., shut down the connections, trace back to identify the intruders, or gather legal evidence to prosecute the intruders, etc.;
- Presenting the traces of the intrusions so that security staff can identify and eliminate the security flaws that have enabled the intrusions.

Intrusion detection assumes that user and program activities in a computer system, intrusive or legitimate, can be monitored. Further, the effects of an intrusion on a system (as monitored and analyzed) are distinct from those of legitimate operations. Many intrusion detection systems, for example IDES [Lunt *et al.*, 1992], utilize audit trails generated by a C2[1] or higher rated computer. Others, for example Bro [Paxson, 1998], monitor network connections and information flows. Different intrusion detection systems use different sets of features (i.e., measures on audit data) and different analytical models to determine whether the system activities are intrusive. Intrusion detection approaches can be categorized into anomaly detection and misuse detection.

2.1 Anomaly Detection

Anomaly detection consists of first establishing the normal behavior profiles for users, programs, or other resources of interest in a system, and observing the actual activities as reported in the audit data to ultimately detect any significant deviations from these profiles. Most anomaly detection approaches are statistical in nature. For example, in SRI's IDES [Lunt *et al.*, 1992] and NIDES [Lunt, 1993], a user's normal profile consists of a set of statistical measures. The measures used in NIDES are of the following types [Lunt, 1993]:

- Ordinal measure: A count of some numerically quantifiable aspect of observed behavior. For example, the amount of CPU time used and the number of audit records produced;

[1]A DoD security classification requiring auditing and unavailability of encrypted passwords.

- Categorical measure: A function of observed behavior over a finite set of categories. Its value is determined by its frequency relative to other categories. It can be further classified as:
  - Binary categorical measure: Whether the category of behavior is present (i.e., 0 or 1). This type of measure is sensitive in detecting infrequently used categories, such as changing one's password;
  - Linear categorical measure: A score function that counts the number of times each category of behavior occurs. For example, command usage is a linear categorical measure, where the categories span all the available command names for that system.

To compute the deviations from the profile, IDES and NIDES use a weighted combining function to sum up the abnormality values of the measures. The profiles are also updated periodically (i.e., aged) based on the (new) observed user behavior to account for normal shifts in user behavior (for example, when a conference deadline approaches).

Anomaly detection systems can detect unknown intrusion since they require no *a priori* knowledge about specific intrusions. Statistical-based approaches also have the added advantage of being adaptive to evolving user and system behavior since updating the statistical measures is relatively easy. However, anomaly detection systems also have major shortcomings:

- The selection of the right set of system (usage) features to be measured can vary greatly among different computing environments;
- The fine tuning of the deviation threshold is very ad hoc;

- User behavior can change dynamically and can be very inconsistent;
- Some intrusions can only be detected by studying the sequential interrelation between events because each event alone can appear to be normal according to the statistical measures.
- A statistical-based system can be trained, over some period of time, by a deliberate intruder to gradually update the user profile to accept his intrusive activities as normal behavior!

2.2 Misuse Detection

Misuse detection consists of first recording and representing the specific patterns of intrusions that exploit known system vulnerabilities or violate system security policies, then monitoring current activities for such patterns, and reporting the matches. There are several approaches in misuse detection. They differ in the representation as well as the matching algorithms employed to detect the intrusion patterns.

Some systems, for example NIDES [Lunt, 1993], use a rule-based expert system component for misuse detection. These systems encode known system vulnerabilities and attack scenarios, as well as intuitions about suspicious behavior, into rules. For example, one such rule is: *more than three consecutive unsuccessful logins within five minutes is a penetration attempt.* Audit data is matched against the rule conditions to determine whether the activities constitute intrusions.

Another system, STAT [Ilgun *et al.*, 1995], uses state transition analysis for misuse detection. It represents and detects known penetration scenarios using state transition diagrams. The intuition behind this approach is that any penetration is essentially a sequence of actions that leads the target system from an initial normal state to a compromised state. Here a state in the state transition diagram is a list of assertions in terms of system attributes and user privileges. A transition is labeled by a user action (i.e., the *signature action*), for example, the acquisition of previously un-held privileges. Intrusions are detected in STAT when a final compromised state in the state transition diagram is reached.

IDIOT [Kumar and Spafford, 1995] uses a more formal pattern classification and matching approach for misuse detection. First, independent of the underlying computational framework of matching, the characteristics of intrusion patterns are partitioned into orthogonal categories:

- *linearity*, which means that the specified sequence of events must occur;
- *unification*, which instantiates variables to earlier events and matches these events to later occurring events, for example, the variable *file2* from different audit records are bound to the same value (a file name) after the unification;
- *occurrence*, which specifies the relative placement in time of an event with respect to the previous events, for example, event *e2* occurs within 5 seconds after event *e1*;
- *beginning*, which specifies the absolute time of the beginning of a pattern;
- *duration*, the time duration for which an event must be active.

Colored Petri Nets were used as the pattern matching model. Each intrusion signature is represented as a Petri net: *linearity* is represented as the sequence of transitions; *unification* is introduced through the use of global variables; and *occurrence*, *beginning* and *duration* are introduced through the use of guard expressions in the Petri nets [Kumar and Spafford, 1995]. A sequence of transitions from the start state(s) to the final state(s) constitutes a match of the intrusion signature.

The key advantage of these misuse detection approaches is that they can accurately and efficiently detect known attacks, those that have been coded as rules or patterns. By their nature, they are not very effective in detecting unknown attacks, those that have no matched rules or patterns, unless the new attacks employ the same system level events manifested by previously encoded exploits. Given the fact that new attack techniques are invented often, misuse intrusion systems may need to be updated frequently across many platforms. However, constructing and maintaining a misuse detection system is very labor-intensive since attack scenarios and system vulnerabilities need to be analyzed and categorized, and the corresponding rules and patterns need to be carefully hand-coded and verified.

2.3 Combining Anomaly and Misuse Detection

Since both anomaly detection and misuse detection have major shortcomings that hamper their effectiveness in detecting certain types of intrusions, many systems employ both approaches. For example, NIDES [Lunt, 1993] has both a statistical-based anomaly detection module and a rule-based expert system for misuse detection. The statistical and rule-based components function in parallel and independently of each other. A separate component, the resolver, is used to filter and combine evidence from the two detection modules to determine a final outcome.

The studies reported in [Jonsson and Olovsson, 1997] suggest that a typical attack session can be split into three phases: a *learning phase*, a *standard attack phase*, and an *innovative attack phase*. During the *learning phase*, an inexperienced attacker learns about the target system's limitations, features and vulnerabilities to prepare himself or herself for the next phase. An experienced attacker directly goes into the *standard attack phase* in which he would try out the known vulnerabilities and attack scripts (e.g., those available on the Web). When all known attack methods fail, the attacker would be forced to enter the *innovative attack phase* and try to discover and exploit vulnerabilities that may be unknown to system administrators of the target system. It is expected that the probability for successful attacks during the *standard attack phase* is considerably high. We can draw from this study the conclusion that a well designed/updated misuse detection module should be used to detect the majority of the attacks, and anomaly detection is the only hope to fight against the innovative and "stealthy" attacks.

2.4 Problems with Current Intrusion Detection Systems

We measure the quality of an IDS by its effectiveness, adaptability and extensibility. An IDS is effective if it has both high intrusion detection (i.e., true positive) rate and low false alarm (i.e., false positive) rate. It is adaptable if it can detect slight variations of the known intrusions, and can be quickly updated to detect new intrusions soon after they are invented. It is extensible if it can incorporate new detection modules or can be customized according to (changed) network system configurations.

Current IDSs lack effectiveness. The hand-crafted rules and patterns, and the statistical measures on selected system measures are the codified “expert knowledge” in security, system design, and the particular intrusion detection approaches in use. Expert knowledge is usually incomplete and imprecise due to the complexities of the network systems.

Current IDSs also lack adaptability. Experts tend to focus on analyzing “current” (i.e., “known”) intrusion methods and system vulnerabilities. As a result, IDSs may not be able to detect “future” (i.e., “unknown”) attacks. Developing and incorporating new detection modules is slow because of the inherent “learning curve”.

Current IDSs lack extensibility. Reuse or customization of an IDS in a new computing environment is difficult because the expert rules and statistical measures are usually ad hoc and environment-specific. Since most current intrusion detection systems are monolithic, it is also hard to add new and complementary detection modules to an existing IDS.

Some of the recent research and commercial IDSs have started to provide built-in mechanisms for customization and extension. For example, both Bro [Paxson, 1998] and NFR [Network Flight Recorder Inc., 1997] filter network traffic streams into a series of events, and execute scripts, e.g., Bro policy scripts and NFR’s N-Codes, that contain site-specific event handlers, i.e., intrusion detection and handling rules. The system administration personnel at each installation site must then assume the roles of both security experts and IDS builders because they are responsible for writing the correct event handling functions. Our first-hand experience with both Bro and NFR show that while these systems provide great flexibility, writing the scripts involves a lot of effort, in addition to learning the scripting languages. For example, there is no means to "debug" the scripts. These systems also handle a fixed set of network traffic event types. On a few occasions we were forced to make changes to the source code of the original IDS to handle new event types.

We can attribute, to a very large extent, the poor qualities of current IDSs to the manual, ad hoc, and purely knowledge engineering development process. Given the complexities of network systems, and the huge amount of audit data generated by user and system activities, we need a more systematic and automatic approach to building IDSs.

2.5 Our Approach

We need a development approach that can meet the requirements (i.e., goals) of IDSs, namely, effective, adaptable, and extensible. We have developed a *systematic framework for mining audit data for automated intrusion detection models*. Specifically, we have designed, developed and made widely available a system, MADAM ID, to assist system administrators or security officers to:

- select appropriate system features for intrusion detection;
- construct inductively learned classifiers as detection models; and
- architect a combined hierarchical detector system from component detectors.

The central theme is to apply well-developed data mining techniques, which are discussed in Chapter 3, to intrusion detection. Using MADAM ID, system features are constructed based on frequent patterns computed from audit data, which capture the actual behavior, in the forms of statistical summaries, of normal activities and intrusions. Therefore, the intrusion detection models that include these constructed features can be more *effective* in distinguishing normal and intrusion activities. Inductive learning algorithms aim to produce "generalizable" models that have good performance on the "unseen" data. Therefore, the inductively learned detection models can be more *adaptive* to variations of known intrusions. MADAM ID also uses a hierarchical architecture of combining multiple detection models to produce *adaptable* and *extensible* IDSs. Each component detection model is specialized to some intrusions (e.g., "new" intrusions), or a specific network component. The component models are "combined" into a meta-detection model through the meta-learning process, which inductively learns how to use the correlations of the predictions of the models to detect intrusions. Adapting or extending an IDS generally involves constructing a new meta-model from a new mix of component models. A key advantage of MADAM ID is that its data mining programs are environment-independent, i.e., they can be applied to any appropriately preprocessed audit data.

It is important to point out that MADAM ID does not completely eliminate the knowledge-engineering elements from the development process. Raw (i.e., binary) audit data needs to be first processed into ASCII forms suitable for data mining tasks. These data preprocessing steps normally require deep domain knowledge. We argue that generic utilities can be developed by network and operating system experts, and made available to all IDSs, as well as other network and system performance analysis systems, as the lowest level building blocks. MADAM ID assumes that such building blocks are available when constructing IDSs. Security experts can also interact with MADAM ID to drive the iterative data mining and feature construction process, and inspect and edit the inductively learned detection rules when needed.

2.5.1 Related Work in Intrusion Detection

In [Forrest *et al.*, 1996], it was shown that the short sequences of system calls made by *sendmail* are very consistent. Hence a database of all short system call sequences from "normal runs" of *sendmail* can be used as a simple and effective anomaly detector. Their findings motivated us to search for simple and accurate intrusion detectors. Chapter 4 details our classification experiments on the same data set reported by [Forrest *et al.*, 1996], which demonstrate that our classifiers are more effective in detecting the anomalies.

A very strongly related work was reported in [Teng *et al.*, 1990]. A time-based inductive engine was used to analyze audit data and generate rule-based sequential patterns that describe user behavior. An example of such rules is $E1 - E2 - E3 \Rightarrow E4 = 95\%$, which indicates that if $E1$, $E2$, and $E3$ occur in serial order then there is 95% chance that $E4$ will follow. The work as described was preliminary, and there have been no follow-up reports. As we discuss in Chapter 5, our frequent episode algorithms can discover more comprehensive sequential (inter-audit record) patterns. In addition, we also consider intra-audit record patterns as part of user or program behavior.

In [Crosbie and Spafford, 1995; Porras and Neumann, 1997], distributed, scalable and cooperative component-based architectures were proposed. The advantages of using such an architecture instead of a monolithic one are: the system can easily be tailored for specific computing environments; the system can be integrated with other complementary detection systems, and components of the system can be deployed in various layers of a network system or different nodes in the network to detect orchestrated attacks. In particular, [Crosbie and Spafford, 1995] proposed an autonomous agent system for intrusion detection. Each agent, using generic algorithms, can learn how to detect anomalous behavior of the system. A group of such agents, each responsible for monitoring part of the network system, works cooperatively to protect the network. MADAM ID supports the construction of distributed and cooperative intrusion detection models because its meta-learning programs can be used to produce a combined detection model that correlates the predictions made by multiple (distributed) detection models.

Chapter 3

Data Mining

Across all industry sectors and scientific research areas, the amount of data collected and warehoused is growing at an explosive rate. However it is believed that less than 10% of the stored data has ever been retrieved and analyzed. The reason is that it is easy and cheap to store the data but difficult and expensive to make good use of the vast amount of data. Since manual approaches are obviously impractical given the sheer volume of data and the demand for fast analysis results, new tools and techniques are emerging to intelligently assist humans in discovering useful knowledge from the database. These techniques and tools are the subject of the growing field of knowledge discovery in databases (KDD) [Fayyad *et al.*, 1996c].

KDD can be defined as "the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data" [Fayyad *et al.*, 1996b]. Data mining is a particular step in this process in which specific algorithms are applied to extract patterns from data [Fayyad *et al.*, 1996c].

3.1 The KDD Process

The KDD process involves a number of steps and is often interactive, iterative, and user-driven [Fayyad *et al.*, 1996c]:

- Getting to know the application domain: trying to understand the data and the discovery task.
- Data preparation: includes creating a target dataset, removing noise from data, and identifying a subset of the variables.
- Data Mining: includes first deciding what model, for example, summarization, classification, or clustering is to be derived from the data; then applying an appropriate algorithm to generate classification rules or trees, association and frequent sequential patterns, etc.
- Interpretation: involves trying to understand the discovered patterns, returning to the previous steps to restart the process using different settings, removing redundant or trivial patterns, and presenting the useful patterns to users.
- Using the discovered knowledge: includes incorporating the knowledge into a production system, or simply reporting it to interested parties.

Data mining is the most critical step in the KDD process. A lot of effort has been devoted to the research and development of general, accurate, and fast data mining algorithms.

3.2 Data Mining Algorithms

There are a wide variety of data mining algorithms, drawn from the fields of statistics, pattern recognition, machine learning, and databases. Several types of algorithms are particularly relevant to intrusion detection:

Classification: classifies a data item into one of several pre-defined categories. These algorithms normally output "classifiers", for example, in the form of decision trees or rules. An ideal application in intrusion detection would be to gather sufficient "normal" and "abnormal" audit data for a user or a program, then apply a classification algorithm to learn a classifier that can label or predict new unseen audit data as belonging to the normal class or the abnormal class;

Link analysis: determines relations between fields in the database records. Correlations of system features in audit data, for example, the correlation between *command* and *argument* in the shell command history data of a user, can serve as the basis for constructing normal usage profiles. A programmer, for example, may have "emacs" highly associated with "C" files;

Sequence analysis: models sequential patterns. These algorithms can discover what time-based sequences of audit events are frequently occurring together. These frequent event patterns provide guidelines for incorporating temporal and statistical measures into intrusion detection models. For example, patterns from audit data containing network-based denial-of-service (DOS) attacks suggest that several per-host and per-service measures should be included (see Chapter 6).

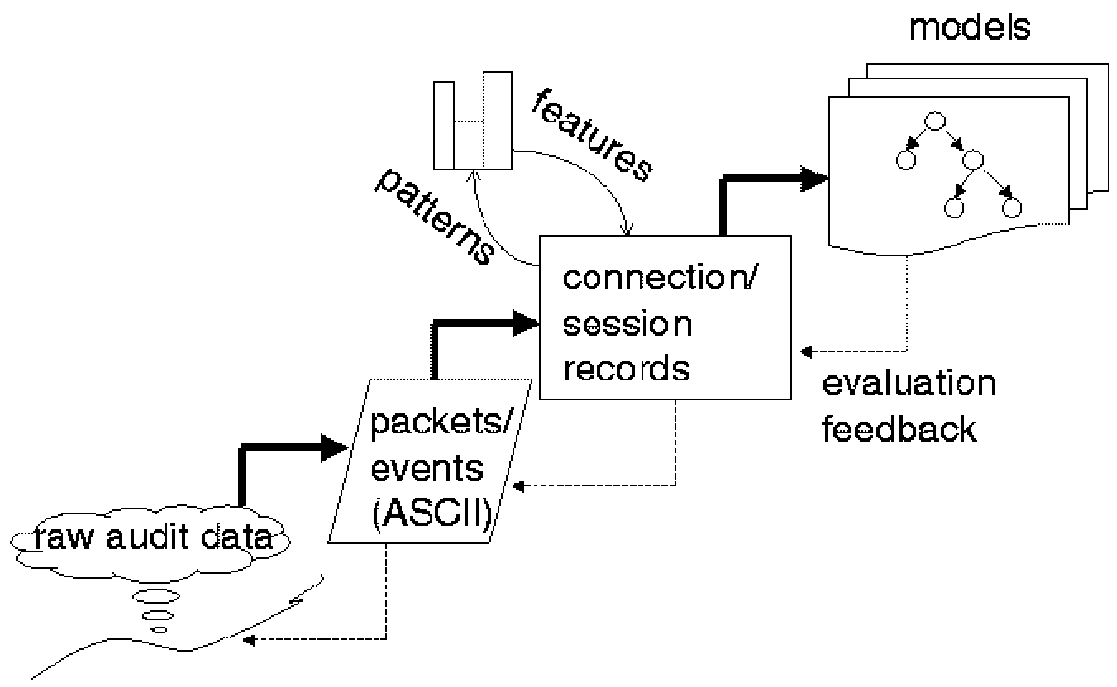


Figure 3.1: The Data Mining Process of Building ID Models

Chapter 4 provides details on how to formulate classification tasks for intrusion detection. Chapter 5 discusses how the association rules algorithm and the frequent episodes algorithm can be applied to audit data.

3.3 The Data Mining Process of Building Intrusion Detection Models

With the recent rapid development in KDD, we have gained a better understanding of the techniques and process frameworks that can support systematic data analysis on the vast amount of audit data (that can be made available). The process of using data mining approaches to build intrusion detection models is shown in Figure 3.1.

Here raw (binary) audit data is first processed into ASCII network packet information (or host event data), which is in turn summarized into connection records (or host session records) containing a number of within-connection features, e.g., *service*, *duration*, *flag* (indicating the normal or error status according to the protocols), etc. Data mining programs are then applied to the connection records to compute the frequent patterns, i.e., association rules and frequent episodes, which are in turn analyzed to construct additional features for the connection records. Classification programs, for example, RIPPER [Cohen, 1995], are then used to inductively learn the detection models. This process is of course iterative. For example, poor performance of the classification models often indicates that more pattern mining and feature construction is needed.

Our research efforts are mainly in: extending the basic association rules and frequent episodes algorithms so that "relevant" patterns can be computed efficiently; automatically constructing features from the mined patterns; and efficient real-time execution of detection models.

3.4 Related Data Mining Applications

Data mining (KDD) techniques have been successfully applied to many business and scientific domains [Brachman *et al.*, 1996; Fayyad *et al.*, 1996a], including some that are closely related to intrusion detection.

Cellular fraud detection is similar to intrusion detection in that unusual behavior is to be distinguished from typical behavior. Historically, manual and ad hoc approaches were used to decide which aspects (features) of customers' behavior should be profiled to construct fraud detectors. In [Fawcett and Provost, 1996], an innovative framework utilizing data mining techniques was proposed to automate the fraud detector construction process. First, a data mining program is used to discover the general patterns (rule sets) of fraudulent usage from a large database of cellular calls. Next, these patterns are used to profile each individual customer's normal usage on an account-day basis. These profiles are statistical measures on the customer's behavior with regard to the patterns. They determine whether the usage behavior is uncharacteristic enough to suggest fraud. Finally, a (learned) detector combines evidence from multiple profiles and generates fraud alarms. It was shown that the prototype system performs nearly as well as the state-of-the-art, hand-tuned system. And it was surprising that some of the features that had been considered to be reliable indicators by experts turned out to contribute little in detecting fraud (in their test data).

A team at Columbia University, headed by Professor Sal Stolfo, is developing JAM, an agent-based distributed data mining framework, and applying it to the problem of credit card fraud detection [Stolfo *et al.*, 1997b]. Here, the main research effort is on using meta-learning to combine multiple base classifiers separately learned from distributed databases. It has been shown that the resulting meta classifier can improve the prediction accuracy over any single base classifier [Stolfo *et al.*, 1997a]. Recent research directions include techniques for combining classifiers that are learned from databases with different schema, and on various methods to boost the accuracy of the meta-classifier. The JAM project directly contributes to MADAM ID, particularly in the mechanisms for combining multiple detection classifiers.

Telecommunication Alarm Sequence Analyzer (TASA) [Hătőnen *et al.*, 1996] is a novel system for alarm management in telecommunication networks. It utilizes data mining algorithms to locate frequently occurring alarm episodes, e.g., *when alarm A and B occur within 5 seconds, alarm C will occur within 60 seconds*, from the alarm stream and present them as rules, which are then integrated into the alarm-handling software of the telephone networks. We implemented a specialized version of the frequent episode algorithm, which is used in TASA, to uncover frequent inter-audit record patterns.

Chapter 4

Classification Models for Intrusion Detection

In this chapter we first give a brief overview of the classification problem, and explain the rationale of using classification rules for intrusion detection. We then describe in detail our experiments in constructing classification rules for anomaly detection. These experiments, first reported in [Lee and Stolfo, 1998], motivated our research in developing MADAM ID. The first set of experiments is on the *sendmail* system call data, and the second is on the network *tcpdump* data. We demonstrate the effectiveness of using classification models for anomaly detection, and discuss the difficulties in building these models.

4.1 Classification

In many applications, e.g., pattern recognition, we need to classify data items into one of a discrete set of possible categories. These *classification* tasks typically require the construction of a *classifier*, a function that assigns a *class label* to each data item described by a set of *attributes*.

More formally, let $\mathcal{A}$ be a vector of attributes $[A_0, A_1, \ldots, A_n]$, where each attribute is either one of two types, discrete or numerical. Let C be the set of class labels. Let $\mathcal{I}$ denote the universe of all possible data items in the problem domain. We write each data item $x \in \mathcal{I}$ as a vector $[v_0, v_1, \ldots, v_n]$ such that each v_i is a value of A_i. Since each data item belongs to a category, it can be assigned a class label. That is, there is a function f from $\mathcal{I}$ to C such that

$$f(x) = c$$

where c is a valid value of C. Function (or model) f is normally unknown, and is the subject of inductive learning.

4.1.1 Learning a Classification Model

Let $\mathcal{D}$ be the given set of pre-labeled *training items*, where each $d_i \in \mathcal{D}$ is a vector $[x_i, c_i]$. Here $x_i \in \mathcal{I}$, and c_i is the known class label of x_i. The goal of the inductive learning process is to construct an *approximation* $\hat{f}$ of f. That is, we want the classifications made by the learned classifier, i.e., $\hat{f}(x_i) = \hat{c}_i$ for each d_i, to agree with the true class labels, i.e., c_i, as much as possible. It is often desirable that $\hat{f}$ has *generalization* ability, that is, the ability to make predictions beyond the observed training data. The *accuracy* of a learned model can therefore be refined into two separate measurements: the *training accuracy* on data it was trained from, and the *generalization accuracy* on unseen data. A set of "set-aside" *testing items*, $\mathcal{T}$, that has the same class distribution as $\mathcal{D}$ and $\mathcal{D} \cap \mathcal{T} = \emptyset$, can be used to estimate the generalization accuracy of the learned model. A model with higher generalization accuracy is normally preferred.

There are several machine learning approaches for computing classification models, for example, decision tree learning [Quinlan, 1986], rule induction [Clark and Niblett, 1989; Cohen, 1995], neural networks [Rumelhart *et al.*, 1994], Bayesian learning [Cheeseman and Stutz, 1996], etc. Each approach uses a different model representation, e.g., a decision tree or a set of rules, etc., and a different search strategy and heuristic for traversing the space of possible models.

Search Heuristic

Information gain is the most important and widely used search heuristic in learning a classification model. In order to understand information gain precisely, we first explain a measure from information theory, *entropy*, that characterizes the "impurity" of a collection of data items. For a dataset $\mathcal{D}$ where each data item belongs to a class $c_i \in C$, the entropy of $\mathcal{D}$ relative to this $|C|$-wise classification is defined as [Mitchell, 1997]

$$Entropy(\mathcal{D}) \equiv \sum_{i=1}^{|C|} -p_i \log p_i \tag{4.1}$$

where p_i is the proportion of $\mathcal{D}$ belonging to class c_i. An interpretation of entropy is that it specifies the number of bits required to encode (and transmit) the classification of a data item. The entropy value is smaller when the class distribution is skewer, i.e., when the data is more "pure". For example, if all data items belong to one class, then entropy is 0, and 0 bit needs to be transmitted because the receiver knows that there is only one outcome. The entropy value is larger when the class distribution is more even, i.e., when the data is more "impure". For example, if the data items are evenly distributed in $|C|$ classes, i.e., $p_i = \frac{1}{|C|}$ for each c_i, then $log|C|$ bits are required to encode a classification.

A classification model is essentially a function, i.e., $\hat{f}$, defined on the attributes of the dataset. Information gain is a measure on the utility of each attribute in classifying the data items. Mitchell [Mitchell, 1997] gives a definition of information gain as

$$Gain(\mathcal{D}, A_i) \equiv Entropy(\mathcal{D}) \; - \sum_{v \in \, Values(A_i)} \frac{\mathcal{D}_v}{\mathcal{D}} Entropy(\mathcal{D}_v) \qquad (4.2)$$

where $Values(A_i)$ is the set of values of attribute A_i, and $\mathcal{D}_v$ is a subset of $\mathcal{D}$ where A_i has value v. The second term in Equation 4.2 is the expected entropy value after $\mathcal{D}$ is partitioned using A_i. $Gain(\mathcal{D}, A_i)$ is thus the expected entropy reduction using attribute A_i, i.e., the number of bits saved when encoding the classification of a data item in $\mathcal{D}$.

The goal of constructing a classification model is that after (selectively) applying a sequence of attribute tests, the dataset can be partitioned into "pure" subsets, i.e., each in a target class. Therefore, the attribute with the largest information gain is considered as the most useful for classifying the examples. Notice that the larger the gain, the smaller the value of the second term in Equation 4.2, which indicates that $\mathcal{D}$ is partitioned by the attribute into class-wise "purer" subsets.

As an example, ID3 [Quinlan, 1986] constructs a decision tree where each node of the tree specifies a test on an attribute, and each branch of the node corresponds to one of its values. The leaves are the classifications. The ID3 algorithm grows a tree, from the root, by selecting from the remaining attributes (with regard to the path of attribute tests from the root to the current node) the one with the highest information gain as the test for the current node.

Inductive Bias

Since there can be a large number of consistent models computed from training data set $\mathcal{D}$, all machine learning algorithms employ some sort of *inductive bias*, that is, preference for one model over another in addition to mere accuracy. Many algorithms use various implementations of the bias

Given two models with the same accuracy, the simpler one should be preferred.

For example, ID3 prefers shorter trees over longer trees. This bias has its origin in (ancient philosopher) William Occam's famous razor [Mitchell, 1997]

Prefer the simplest hypothesis that fits the data.

It was pointed out that this Occam's razor bias should only be applied when comparing models using the generalization accuracy, but not the training accuracy [Domingos, 1998].

4.1.2 Classification Rules

In many applications, it is desirable to learn a set of propositional if-then rules that jointly define the target function. Rule sets are relatively easy for humans to understand, and can easily be incorporated into (existing) general rule execution engines. One solution is to learn a model, e.g., a decision tree, and then translate the model into an equivalent set of rules, e.g., one rule for each leaf node in the tree. A better solution is to directly apply a rule learning algorithm. Rule learners can outperform decision tree learners and other algorithms on many problems, and can easily incorporate certain types of prior knowledge into the learning process [Cohen, 1994].

RIPPER [Cohen, 1995] is the rule learning algorithm used in our study. Each RIPPER rule consists of a conjunction of conditions, i.e., attribute-value tests, and a consequence, i.e., a class label. A condition is of the form $A_d = v$, $A_n \leq \theta$ or $A_n \geq \theta$, where A_d is a discrete attribute and v a legal value of A_d, or A_n is a numerical attribute and θ is some value of A_n. A data item is *covered* by a rule if its attribute values satisfy the conditions of the rule. To a rule's target class, i.e., its consequence, data items that belong to this class are called the *positive* examples, and those in other classes are called the *negative* examples.

A RIPPER rule is learned in two phases, a *growing* phase and a *pruning* phase. The training data set $\mathcal{D}$ is also partitioned into a growing set and a pruning set for these two phases. A rule is "grown", from an empty conjunction, by repeatedly adding a condition that maximizes the *FOIL information gain* criterion [Quinlan, 1990] until the rule covers no negative data items in the the growing data set. The FOIL information gain is measured based on the number of positive and negative examples covered before and after adding the condition to the rule. It can be interpreted as the reduction in the total number of bits required to encode the classification of all positive examples covered by the rule [Mitchell, 1997]. After the growing phase, a rule is immediately pruned, i.e., simplified, by repeatedly deleting a condition (or a conjunction of conditions) that can lead to a more accurate rule on the pruning data set. The pruning stage is thus a means to improve both the generalization accuracy and the simplicity of the rule.

RIPPER can handle multiple classes. Given training data set $\mathcal{D}$ with n classes. The algorithm first orders the classes in increasing order of prevalence. That is, among the ordered $c_1, c_2, \ldots, c_n$, c_1 is the least prevalent class and c_n is the most prevalent. RIPPER first learns a rule set that separates c_1 from the rest of the classes, removes from $\mathcal{D}$ the data items covered by the learned rules, and then learns another rule set that separates c_2 from $c_3, c_4, \ldots, c_n$. This process continues until there is a single class c_n, which is called the *default class*.

4.1.3 Classification Rules as Intrusion Detection Models

MADAM ID produces classification rules, i.e., RIPPER rules, as intrusion detection models. Since these learned rules have the standard if-then format, with minimum processing, they can be used in many rule-based IDSs in the same manner as hand-coded rules. Further, the RIPPER rules are concise and intuitive, and can be inspected and edited by security experts when needed. That is, they can be checked for "sanity" before being employed in the actual systems.

RIPPER rules have two very desirable properties for intrusion detection: good generalization accuracy and concise (simplified) conditions. There will always be "new" intrusions, in the forms of slight variations of "known" intrusions or completely new breed of attacks, after the intrusion detection models are built. The ability to detect these new intrusions, that is, the generalization accuracy of the rules on the unseen data, is therefore critical for an IDS. Real-time IDSs require simple models for the sake of efficiency. Rule execution entails computing and testing the attribute values for rule condition checking. Having concise conditions therefore contributes to the real-time performance of IDSs.

Why not Alternative Models?

In this thesis study, we did not test the feasibility and effectiveness of using alternative learning techniques, e.g., Hidden Markov Models (HMMs), neural networks, etc., to construct intrusion detection models. We believe that, at this stage of our research, the essential task is to develop an automatic approach for formulating machine learning tasks, i.e., constructing dataset $\mathcal{I}$ from the unstructured raw audit data stream. Regardless of the specific machine learning algorithm to be used, we need to supply it with data items that bear "learnable" characteristics expressed in a set of meaningful attributes. In machine learning terms, we need to first deal with the *feature construction* problem. Therefore, using a "sensible" model, i.e., classification rules, suffices to evaluate the concepts and algorithms in MADAM ID.

An attribute is usually referred to as a *feature* when it is used in a learned model. The feature construction problem arises when given unstructured data, where a data item cannot be expressed as a vector of attribute values because the attributes are not yet "defined". It is sometimes desirable to construct additional features even if a basic set of features is in place. For example, a new feature that computes an aggregate function over some other existing features may be more useful in predicting a certain class.

From the discussions in Sections 4.1.1 and 4.1.2, we know that machine learning algorithms search for attribute (feature) value tests according to their information gain measures, i.e., how well an attribute separates data items from different classes. The goal of feature construction is therefore to add features that have large information gains. That is, the added features will be selected by the learning algorithms, and resulted classification models will be more accurate.

System audit data is available as "raw", i.e., unstructured data. Data preprocessing is needed before intrusion detection analysis can take place. In the context of MADAM ID, we can regard audit data preprocessing as a transformation function T from raw dataset $\mathcal{R}$ to machine learning dataset $\mathcal{I}$, where for each $r_i \in \mathcal{R}$,

$$T(r_i) = x_i \in \mathcal{I}$$

and x_i is an attribute-value vector $[v_0, v_1, \ldots, v_n]$. T essentially defines the set of attributes on $\mathcal{I}$. A poor T can result in attributes that have very low information gains, i.e., the same attribute value occurs evenly in data items of different classes. In extreme cases, we can have $T(r_i) = T(r_j)$ where $f(r_i) \neq f(r_j)$. These data items that share the same attribute values but are in different classes are regarded as "noise". It is extremely difficult to learn an accurate model given very noisy data. Our goal is to provide tools to improve and semi-automate T so that accurate models can be learned from $\mathcal{I}$.

In short, our research deals with the more important and urgent feature selection and construction problems. In our future work, we plan to evaluate the effectiveness and efficiency of alternative machine learning models. The experiments described in the rest of the chapter demonstrate the effectiveness of classification models, and illustrate the importance of feature construction.

4.2 Experiments on *sendmail* Data

There have been a lot of attacks on computer systems that are carried out as exploitations of the design and programming errors in privileged programs, those that can run as root. For example, a flaw in the *finger* daemon allows the attacker to use "buffer overflow" to trick the program to execute his malicious code. Recent research efforts by Ko et al. [Ko *et al.*, 1994] and Forrest et al. [Forrest *et al.*, 1996] attempted to build intrusion detection systems that monitor the execution of privileged programs and detect the attacks on their vulnerabilities. Ko et al. used a program policy specification language to codify the access rights for the privileged programs. Their studies showed that these program policies can be used to detect known exploitations. The approach taken by Forrest et al. originated from their research in computer immune systems. They discovered that the short sequences of system calls made by a program during its normal executions are very consistent. More importantly, the sequences are different from the sequences of its abnormal (exploited) executions as well as the executions of other programs. Therefore a database containing these normal sequences can be used as the "self" definition of the normal behavior of a program, and as the basis to detect anomalies.

Stephanie Forrest provided us with a set of traces of the *sendmail* program used in her experiments [Forrest *et al.*, 1996]. We conducted two sets of experiments. First, sequences of n consecutive system calls were extracted from the *sendmail* traces and supplied to a machine learning algorithm to learn the patterns of "normal" and "abnormal" sequences. These patterns can then be used to examine a new trace and determine whether it contains sufficient abnormal sequences to be identified as an intrusion (anomaly). In the second set of experiments, the goal of the machine learning tasks was to generate rules that predict: 1) the nth system call given the $n-1$ preceding system calls; 2) the middle system call in a sequence of n system calls. These rules of normal *sendmail* system calls can be used to analyze a new trace to detect violations, that is, system calls appearing "out of place". A large number of violations suggest an intrusion. In this section we detail the methods and results of these experiments.

4.2.1 The *sendmail* System Call Traces

The procedure of generating the *sendmail* traces were detailed in [Forrest *et al.*, 1996]. Briefly, each file of the trace data has two columns of integers, the first is the process ids and the second is the system call "numbers". These numbers are indices into a lookup table of system call names. For example, the number "5" represents system call *open*. The set of traces include:

- Normal traces: a trace of the *sendmail* daemon and a concatenation of several invocations of the *sendmail* program;

- Abnormal traces: 3 traces of the *sscp* (*sunsendmailcp*) attacks, 2 traces of the *syslog-remote* attacks, 2 traces of the *syslog-local* attacks, 2 traces of the *decode* attacks, 1 trace of the *sm5x* attack and 1 trace of the *sm565a* attack. These are the traces of (various kinds of) abnormal runs of the *sendmail* program.

4.2.2 Learning the Patterns of "Normal" and "Abnormal" Sequences

In order for a machine learning program to learn the patterns of the "normal" and "abnormal" sequences of system calls, we need to supply it with a set of training data containing pre-labeled "normal" and "abnormal" sequences. We use a sliding window to scan the normal traces and create a list of unique sequences of system calls. We call this list the "normal" list. Next, we scan each of the intrusion traces. For each sequence of system calls, we first look it up in the normal list. If an exact match can be found then the sequence is labeled as "normal". Otherwise it is labeled as "abnormal" (note that the exhaustive data gathering process described in [Forrest *et al.*, 1996] ensured that the normal traces include nearly all possible "normal" short sequences of system calls, as new runs of *sendmail* failed to generate new sequences). Needless to say, all sequences in the normal traces are labeled as "normal". See Table 4.1 for an example of the labeled sequences. It should be noted that an intrusion trace contains many normal sequences in addition to the abnormal sequences since the illegal activities only occur in some places within a trace.

System Call Sequences (length 7)	Class Labels
4 2 66 66 4 138 66	"normal"
...	...
5 5 5 4 59 105 104	"abnormal"
...	...

Table 4.1: Pre-labeled System Call Sequences of Length 7

The Machine Learning Tasks

We applied RIPPER to our training data. The following learning tasks were formulated to induce the rule sets for normal and abnormal system call sequences:

- Each record has n positional attributes, p_1, p_2, $\ldots$, p_n, one for each of the system calls in a sequence of length n; plus a class label, "normal" or "abnormal"

- The training data is composed of normal sequences taken from 80% of the normal traces, plus the abnormal sequences from 2 traces of the *sscp* attacks, 1 trace of the *syslog-local* attack, and 1 trace of the *syslog-remote* attack

- The testing data includes both normal and abnormal traces not used in the training data.

RIPPER outputs a set of if-then rules for the "minority" classes, and a default "true" rule for the remaining class. Table 4.2 shows exemplar RIPPER rules generated from the system call sequence data.

RIPPER rule	Meaning
normal:- $p_2 = 104$, $p_7 = 112$.	If p_2 is 104 (*vtimes*) and p_7 is 112 (*vtrace*) then the sequence is "normal".
normal:- $p_6 = 19$, $p_7 = 105$.	If p_6 is 19 (*lseek*) and p_7 is 105 (*sigvec*) then the sequence is "normal".
...	...
abnormal:- true.	If none of the above, the sequence is "abnormal".

Table 4.2: Example RIPPER Rules for Classifying System Call Sequences

These RIPPER rules can be used to predict whether a sequence is "abnormal" or "normal". But what the intrusion detection system needs to know is whether the trace being analyzed is an intrusion or not. Can we say that whenever there is a predicted abnormal sequence in the trace, it is an intrusion? It depends on the accuracy of the rules when classifying a sequence as abnormal. Unless it is close to 100%, it is unlikely that a predicted abnormal sequence is always part of an intrusion trace, since it can just be an error of the RIPPER rules.

Post-processing for Intrusion Detection

We use the following post-processing scheme to detect whether a given trace is an intrusion based on the RIPPER predictions of its constituent sequences:

1. Use a sliding window of length $2l + 1$, e.g., 7, 9, 11, 13, etc., and a sliding (shift) step of l, to scan the predictions made by the RIPPER rules on system call sequences. Note that l is an input parameter.

2. For each of the (length $2l + 1$) regions of RIPPER predictions generated in Step 1, if more than l predictions are "abnormal" then the current region of predictions is an "abnormal" region.

3. If the percentage of abnormal regions is above a threshold value, say 2%, then the trace is an intrusion.

This scheme is an attempt to filter out the spurious prediction errors. The intuition behind it is that when an intrusion actually occurs, the majority of adjacent system call sequences are abnormal; whereas the prediction errors tend to be isolated and sparse. In [Forrest *et al.*, 1996], the percentage of the mismatched sequences (out of the total number of matches (lookups) performed for the trace) is used to distinguish normal from abnormal. The "mismatched" sequences are the abnormal sequences in our context. Our scheme is different in that we look for abnormal regions that contain more abnormal sequences than the normal ones, and calculate the percentage of abnormal regions (out of the total number of regions). Our scheme is more sensitive to the temporal information, and is less sensitive to noise (i.e., classification errors).

Results and Analysis

Recall that RIPPER only outputs rules for the "minority" class. For example, in our experiments, if the training data has fewer abnormal sequences than the normal ones, the output RIPPER rules can be used to identify abnormal sequences, and the default (i.e., the case of "everything else") prediction is normal. We conjectured that a set of specific rules for normal sequences can be used as the "identity" of a program, and thus can be used to detect any known and unknown intrusions. That is, the rules can be used for anomaly detection. Whereas having only the rules for abnormal sequences only gives us the capability to identify known intrusions. That is, they can be used for misuse detection.

We compare the results of the following experiments that have different distributions of abnormal versus normal sequences in the training data:

1. Experiment A: 46% normal and 54% abnormal. Sequence length is 11.
2. Experiment B: 46% normal and 54% abnormal. Sequence length is 7.
3. Experiment C: 46% abnormal and 54% normal. Sequence length is 11.
4. Experiment D: 46% abnormal and 54% normal. Sequence length is 7.

Table 4.3 shows the results of using the classifiers from these experiments to analyze the traces. We report here the percentages of abnormal regions, as measured by our post-processing scheme, of each trace, and compare our results with Forrest et al., as reported in [Forrest *et al.*, 1996]. From Table 4.3, we can see that in general, intrusion traces generate much larger percentages of abnormal regions than the normal traces. We call these measured percentages the "scores" of the

	% abnormal sequences	% abnormal regions			
Traces	Forrest et al.	Exp. A	Exp. B	Exp. C	Exp. D
sscp-1	5.2	41.9	32.2	40.0	33.1
sscp-2	5.2	40.4	30.4	37.6	33.3
sscp-3	5.2	40.4	30.4	37.6	33.3
syslog-r-1	5.1	30.8	21.2	30.3	21.9
syslog-r-2	1.7	27.1	15.6	26.8	16.5
syslog-l-1	4.0	16.7	11.1	17.0	13.0
syslog-l-2	5.3	19.9	15.9	19.8	15.9
decode-1	0.3	4.7	2.1	3.1	2.1
decode-2	0.3	4.4	2.0	2.5	2.2
sm565a	0.6	11.7	8.0	1.1	1.0
sm5x	2.7	17.7	6.5	5.0	3.0
sendmail	0	1.0	0.1	0.2	0.3

Table 4.3: Comparing Detection of Anomalies. The column Forrest et al. is the percentage of the abnormal sequences of the traces. Columns A, B, C, and D are the percentages of abnormal regions (as measured by the post-processing scheme) of the traces. *sendmail* is the 20% normal traces not used in the training data. Traces in bold were included in the training data, the other traces were used as testing data only.

traces. In order to establish a threshold score for identifying intrusion traces, it is desirable that there is a sufficiently large gap between the scores of the normal sendmail traces and the low-end scores of the intrusion traces. Comparing experiments that used the same sequence length, we observe that such a gap in A, 3.4, is larger than the gap in C, 0.9; and 1.9 in B is larger than 0.7 in D. The RIPPER rules from experiments A and B describe the patterns of the normal sequences. Here the results show that these rules can be used to identify the intrusion traces, including those not seen in the training data, namely, the *decode* traces, the *sm565a* and *sm5x* traces. This confirms our conjecture that rules for normal patterns can be used for anomaly detection. The RIPPER rules from experiments C and D specify the patterns of abnormal sequences in the intrusion traces included in the training data. The results indicate that these rules are very capable of detecting the intrusion traces of the "known" types, i.e., those seen in the training data, namely, the *sscp-3* trace, the *syslog-remote-2* trace and the *syslog-local-2* trace. But compared with the rules from A and B, the rules in C and D perform less effectively on intrusion traces of "unknown" types. This confirms our conjecture that rules for abnormal patterns are good for misuse intrusion detection, but may not be as effective in detecting future ("unknown") intrusions.

The results from Forrest et al. showed that their method required a very low threshold in order to correctly detect the *decode* and *sm565a* intrusions. While the results here show that our approach generated much stronger "signals" of anomalies from the intrusion traces, it should be noted that their method used all of the normal traces but not any of the intrusion traces in training.

4.2.3 Learning to Predict System Calls

Unlike the first set of experiments, which required abnormal traces in the training data, for the second set of experiments we wanted to study how to compute an anomaly detector given just the normal traces. We conducted experiments to learn the normal correlation among system calls: the nth system calls or the middle system calls in normal sequences of length n.

The learning tasks were formulated as follows:

- Each record has $n-1$ positional attributes, p_1, p_2, ... , p_{n-1}, each being a system call; plus a class label, the system call of the nth position or the middle position

- The training data is composed of normal sequences taken from 80% of the normal sendmail traces

- The testing data is the traces not included in the training data, namely, the remaining 20% of the normal sendmail traces and all the intrusion traces.

Table 4.4 shows exemplar RIPPER rules for predicting system calls.

RIPPER rule	Meaning
38:- $p_3 = 40$, $p_4 = 4$.	If p_3 is 40 (*lstat*) and p_4 is 4 (*write*), then the 7th system call is 38 (*stat*).
102:- $p_1 = 106$, $p_4 = 101$.	If p_1 is 106 (*sigblock*) and p_4 is 101 (*bind*), then the 7th system call is 102 (*setsockopt*).
...	...
5:- true.	If none of the above, then the 7th system calls is 5 (*open*).

Table 4.4: Example RIPPER Rules for Classifying System Calls

Each of these RIPPER rules has some “confidence” information: the number of matched examples (records that conform to the rule) and the number of unmatched examples (records that are in conflict with the rule) in the training data. For example, the rule for “38 (*stat*)” covers 12 matched examples and 0 unmatched examples. We measure the confidence value of a rule as the number of matched examples divided by the sum of matched and unmatched examples. These rules can be used to analyze a trace by examining each sequence of the trace. If a violation occurs, i.e., the actual system call is not the same as the one predicted by the rule, the “score” of the trace is incremented by 100 times the confidence of the violated rule. For example, if a sequence in the trace has $p_3 = 40$ and $p_4 = 4$, but $p_7 = 44$ instead of 38, the total score of the trace is incremented by 100 since the confidence value of this violated rule is 1. The averaged score (by the total number of sequences) of the trace is then used to decide whether an intrusion has occurred.

	averaged score of violations			
Traces	Exp. A	Exp. B	Exp. C	Exp. D
sscp-1	24.1	13.5	14.3	24.7
sscp-2	23.5	13.6	13.9	24.4
sscp-3	23.5	13.6	13.9	24.4
syslog-r-1	19.3	11.5	13.9	24.0
syslog-r-2	15.9	8.4	10.9	23.0
syslog-l-1	13.4	6.1	7.2	19.0
syslog-l-2	15.2	8.0	9.0	20.2
decode-1	9.4	3.9	2.4	11.3
decode-2	9.6	4.2	2.8	11.5
sm565a	14.4	8.1	9.4	20.6
sm5x	17.2	8.2	10.1	18.0
sendmail	5.7	0.6	1.2	12.6

Table 4.5: Detecting Anomalies Using Predicted System Calls. Columns A, B, C, and D are the averaged scores of violations of the traces. *sendmail* is the 20% normal traces not used in the training data. None of the intrusion traces was used in training.

Table 4.5 shows the results of the following experiments:

1. Experiment A: predict the 11th system call.

2. Experiment B: predict the middle system call in a sequence of length 7.

3. Experiment C: predict the middle system call in a sequence of length 11.

4. Experiment D: predict the 7th system call.

We can see from Table 4.5 that the RIPPER rules from experiments A and B are effective because the gap between the score of normal sendmail and the low-end scores of intrusion traces, 3.7 and 3.3, respectively, are large enough. However, the rules from C and D perform poorly. Since C predicts the middle system call of a sequence of length 11 and D predicts the 7th system call, we reason that the training data (i.e., the normal traces) has no stable patterns for the 6th or 7th position in system call sequences.

4.2.4 Discussion

Our experiments showed that the normal behavior of a program execution can be established and used to detect its anomalous usage. This confirms the results of other related work in anomaly detection. The weakness of the model in [Forrest *et al.*, 1996] may be that the recorded (rote learned) normal sequence database may be too specific as it contains $\sim 1,500$ entries. Here we show that a machine learning program, RIPPER, was able to generalize the system call sequence information, from 80% of the normal sequences, to a set of concise and accurate rules. The rule sets from these experiments have 200 to 280 rules, and each rule has 2 or 3 attribute tests. We demonstrated that these rules were able to identify previously unseen intrusion traces as well as normal traces.

We need to search for a more predictive classification model so that the anomaly detector has higher confidence in flagging intrusions. Improvement in accuracy can come from adding more features, rather than just the system calls, into the models of program execution. For example, the directories and the names of the files touched by a program can be used. In [Frank, 1994], it is reported that as the number of features increases from 1 to 3, the classification error rate of their network intrusion detection system decreases dramatically. Furthermore, the error rate stabilizes after the size of the feature set reaches 4, the optimal size in their experiments. Many operating systems provide auditing utilities, such as the BSM audit of Solaris, that can be configured to collect abundant information (i.e., with many features) of the activities in a host system. From the audit trails, information about a program or a user can then be extracted. The challenge now is to efficiently compute accurate patterns from the audit data for programs and users.

A key assumption in using a learning algorithm for anomaly detection, and to some degree, misuse detection, is that the training data is nearly “complete” with regard to all possible “normal” behavior of a program or user. Otherwise, the learned detection model can not confidently classify or label an unmatched data as “abnormal” since it can just be an unseen “normal” data. For example, the experiments in Section 4.2.3 used 80% of “normal” system call sequences; whereas the experiments in Section 4.2.2 actually required all “normal” sequences in order to pre-label the “abnormal” sequences to create the training data. During the audit data gathering process, we want to ensure that as much different normal behavior as possible is captured. We first need to have a simple and incremental summary measure of an audit trail so that we can update this measure as each new audit trail is processed, and can stop the audit process when the measure stabilizes. In Chapter 5, we propose to use the frequent intra- and inter- audit record patterns as the summary measure of an audit trail, and describe the algorithms to compute these patterns.

4.3 Experiments on *tcpdump* Data

There are two approaches for network intrusion detection: one is to analyze the audit data on each host of the network and correlate the evidence from the hosts. The other is to monitor the network traffic directly using a packet capturing program such as *tcpdump* [Jacobson *et al.*, 1989]. In this section, we describe how classifiers can be induced from *tcpdump* data to distinguish network attacks from normal traffic.

4.3.1 The *tcpdump* Data

We obtained a set of *tcpdump* data, available at `http://iris.cs.uml.edu:8080/network.html`, that is part of an Information Exploration Shootout (see `http://iris.cs.uml.edu:8080`). The *tcpdump* was executed on the gateway that connects the enterprise LAN and the external networks. It captured the headers (not the user data) of the network packets that passed by the network interface of the gateway. Network traffic between the enterprise LAN and external networks, as well as the broadcast packets within the LAN were therefore collected. For the purposes of the shootout, filters were used so that *tcpdump* only collected Internet Transmission Control Protocol (TCP) and Internet User Datagram Protocol (UDP) packets. The data set consists of 3 runs of *tcpdump* on generated network intrusions and one *tcpdump* run on normal network traffic (with no intrusions). The output of each *tcpdump* run is in a separate file. The traffic volume (number of network connections) of these runs are about the same. Our experiments focused on building an anomaly detection model from the normal dataset.

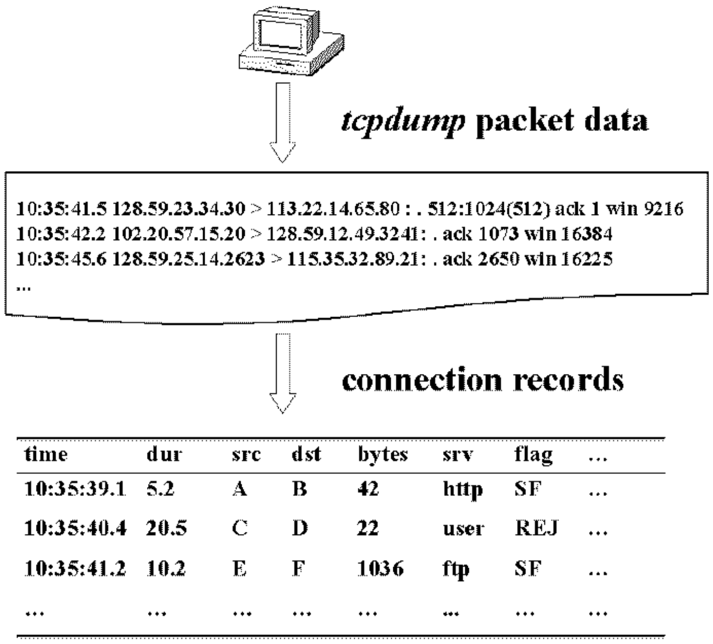
Since *tcpdump* output is not intended specifically for security purposes, we had to go through multiple iterations of data pre-processing to extract meaningful features and measures. We studied TCP/IP and its security related problems, for example [Stevens, 1994; Paxson, 1997; Atkins *et al.*, 1996; Paxson, 1998; Bellovin, 1989; Porras and Valdes, 1998], for guidelines on the protocols and the important features that characterize a connection.

4.3.2 Data Pre-processing

Our goal is to build a model that can predict a network connection as normal or an intrusion. *tcpdump* output contains packet data of network connections, in the order of packet appearances in the network. Packet data therefore needs to be preprocessed in order to gather connection-level information. Figure 4.1 depicts the data preprocessing process. Here, binary *tcpdump* data is first converted into ASCII packet level data, where each line contains the information of one network packet. The data is ordered by the timestamps of the packets. Therefore, packets belonging to different connections may be interleaved. For example, the 3 packets shown in the figure are from different connections. The packet data is then processed into connection records with a number of features (i.e. attributes), e.g., *time* (the starting time of the connection, i.e., the timestamp of its first packet), *dur* (the duration of the connection), *src* and *dst* (source and destination hosts), *bytes* (number of data bytes from source to destination), *srv* (the service, i.e., port, in the destination), and *flag* (how the connection conforms to the network protocols, e.g., *SF* is normal, *REJ* is "rejected"), etc. These intrinsic features essentially summarize the packet level information within a connection.

We developed a script to scan each *tcpdump* ASCII packet data file and extract the "connection-level" information to form connection records. For each TCP connection, the script processes packets between the two ports of the participating hosts, and:

- checks whether 3-way handshake has been properly followed to establish the



```
10:35:41.5 128.59.23.34.30 > 113.22.14.65.80 : . 512:1024(512) ack 1 win 9216
10:35:42.2 102.20.57.15.20 > 128.59.12.49.3241: . ack 1073 win 16384
10:35:45.6 128.59.25.14.2623 > 115.35.32.89.21: . ack 2650 win 16225
...
```

time	dur	src	dst	bytes	srv	flag	...
10:35:39.1	5.2	A	B	42	http	SF	...
10:35:40.4	20.5	C	D	22	user	REJ	...
10:35:41.2	10.2	E	F	1036	ftp	SF	...
...	...	...	...	...	...	...	...

Figure 4.1: Processing Packet-level Network Audit Data into Connection Records

connection. The following errors are recorded: connection rejected, connection attempted but not established (i.e., the initiating host never receives a SYN acknowledgment), and unwanted SYN acknowledgment received (i.e., no connection request, a SYN packet, was sent first),

- monitors each data packet and ACK packet, keeps a number of counters in order to calculate these statistics of the connection: resent rate, wrong resent rate, duplicate ACK rate, hole rate, wrong data packet size rate, total number of data bytes sent in each direction, percentage of data packets, and percentage of control packets, and

- watches how the connection is terminated: normal (i.e., both sides properly send and receive FINs), abort (i.e., one host sends RST to terminate, and all data packets are properly ACKed), half closed (i.e., only one host sends FIN), and disconnected.

Since UDP is connectionless (i.e., no connection state), we simply treat each packet as a connection.

A connection record, in preparation of machine learning, now has the following attributes (features): start time, duration, participating hosts, ports, the statistics of the connection (e.g., bytes sent in each direction, resent rate, etc.), flag (i.e., "normal" or one of the recorded connection/termination errors), and protocol type (i.e., TCP or UDP). From the ports, we know whether the connection is to a well-known service, e.g., *http* (port 80), or to a user application.

We call the host that initiates the connection, i.e., the one that sends the first SYN, as the source, and the other as the destination. Depending on the direction from the source to the destination, a connection is one of the three types: *out-going* - from the LAN to the external networks; *in-coming* - from the external networks to the LAN; and *inter-LAN* - within the LAN. Taking the topology of the network into consideration is important in network intrusion detection. Intuitively, intrusions, which come from outside, may first exhibit some abnormal patterns (e.g., penetration attempts) in the *in-coming* connections, and subsequently in the *inter-LAN* (e.g., doing damage to the LAN) and/or the *out-going* (e.g., stealing/uploading data) connections. Analyzing these types of connections and constructing corresponding detection models separately may improve detection accuracy.

4.3.3 Experiments and Results

For each type (i.e., direction) of the connections, we formulated the classification experiments as the following:

- Each connection record uses the destination service (i.e., destination port) as the class label, and all the other connection features as attributes;
- The training data is 80% of the connections from the normal *tcpdump* data file, while the test data includes the remaining 20% from the normal *tcpdump* data file, and all the connections from the 3 *tcpdump* data files marked as having embedded attacks;
- 5-fold cross validation evaluation is reported here. The process of training and testing is repeated 5 times, each time using a different 80% of the normal data as the training data, and accordingly the different remaining 20% of the normal data as part of the test data. The averaged accuracy of the classifiers from the 5 runs is reported.

We again applied RIPPER to the connection data. The resulting classifier characterizes the normal patterns of each service in terms of the connection features. When using the classifier on the testing data, the percentage of misclassifications on each *tcpdump* data set is reported. Here a misclassification is the situation where the classifier predicts a destination service, according to the connection features, that is different from the actual. This misclassification rate should be very low for normal connection data and high for intrusion data. The intuition behind this classification model is straightforward: when intrusions take place, the features (i.e.,

	% misclassification (by traffic type)		
Connection Data	out-going	in-coming	inter-LAN
normal	3.91%	4.68%	4%
intrusion1	3.81%	6.76%	22.65%
intrusion2	4.76%	7.47%	8.7%
intrusion3	3.71%	13.7%	7.86%

Table 4.6: Misclassification Rate on Normal and Intrusion Data. Separate classifiers were trained and tested on connection data of each traffic type. "normal" is the 20% data set aside from the training data. No intrusion data was used for training.

characteristics) of connections to certain services, for example, *ftp*, are different from the normal traffic patterns of the same service.

The results from the first round of experiments, as shown in Table 4.6, were not very good: the differences in the misclassification rates of the normal and intrusion data were small, except for the *inter-LAN* traffic of some intrusions.

We then redesigned our set of features by adding some continuous and intensity measures into each connection record:

- Examining all connections in the past n seconds, and counting the number of: connection establishment errors (e.g., "connection rejected"), all other types of errors (e.g., "disconnected"), connections to designated system services (e.g., *ftp*), connections to user applications, and connections to the same service as the current connection;

- Calculate for the past n seconds, the per-connection average duration and data bytes (on both directions) of all connections, and the same averages of connections to the same service.

These temporal and statistical features provide additional information of the network activity from a continuous perspective, and provide more insight into

	% misclassification (by traffic type)		
Connection Data	out-going	in-coming	inter-LAN
normal	0.88%	0.31%	1.43%
intrusion1	2.54%	27.37%	20.48%
intrusion2	3.04%	27.42%	5.63%
intrusion3	2.32%	42.20%	6.80%

Table 4.7: Using Temporal and Statistical Measures to Improve Classification Accuracy. Here the time interval is 30 seconds.

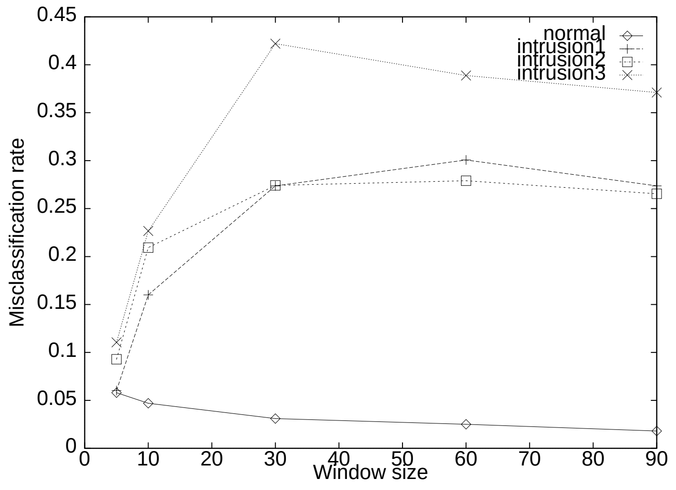


Figure 4.2: Effects of Window Sizes on Misclassification Rates

anomalies. For example, a low rate of error due to innocent attempts and network glitches in a short time span is expected, but an excess beyond the averaged norm indicates anomalous activity. Table 4.7 shows the improvement of adding these features. Here, using a time interval of 30 seconds (i.e., $n = 30s$), we see that the misclassification rates on the intrusion data are much higher than the normal data, especially for the *in-coming* traffic. The RIPPER rule set (i.e., the classifier) has just 9 rules and 25 conditions. For example, one rule says "if the average number of bytes from source to destination of the connections to the same service is 0, and the percentage of control packets in the current connection is 100%, then the service is *auth*".

To understand the effects of the time intervals on the misclassification rates, we ran the experiments using various time intervals: 5s, 10s, 30s, 60s, and 90s. The effects on the *out-going* and *inter-LAN* traffic were very small. However, as Figure 4.2 shows, for the *in-coming* traffic, the misclassification rates on the intrusion data increase dramatically as the time interval goes from 5s to 30s, then stabilizes or tapers off afterwards.

4.3.4 Discussion

We learned some important lessons from the experiments on the *tcpdump* data. First, when the collected data is not designed specifically for security purposes or can not be used directly to build a detection model, a considerable amount of iterative data pre-processing is required. This process requires a lot of domain knowledge, and may not be easily automated. Second, in general, adding temporal and statistical features can improve the accuracy of the classification model for network traffic.

There are also much needed improvements to our current approach: First, deciding upon the right set of features is difficult and time consuming. For example, many trials were attempted before we came up with the current set of features and time intervals. We need useful tools that can provide insight into the behavior patterns that may be exhibited in the data. Second, we should provide tools that can help administrative staff understand the nature of the anomalies. Again, we need tools that can compare the anomalous traffic against the normal historical traffic, in terms of easy-to-understand patterns.

4.4 Summary

In this chapter, we briefly reviewed the problem of building classification models, and discussed the rationale of using classification rules as intrusion detection models. Through experiments on *sendmail* and *tcpdump* data, we demonstrated the effectiveness of using classification models for intrusion detection. We showed that the classification rules are accurate, concise, and intuitive.

In order to construct an accurate classifier, we need to gather a sufficient amount of training data and identify a set of meaningful features. Both of these tasks require insight into the data, and can be difficult without proper tools and guidelines. In Chapter 5, we propose some algorithms that will address these needs.

Chapter 5

Algorithms for Mining Audit Data

In this chapter we first give an in-depth analysis of the requirements for building effective classification rules for intrusion detection, namely, automatic tools are needed to guide audit data gathering and feature construction. We argue that frequent patterns, i.e., the association rules and frequent episodes, from audit records can serve these purposes. We give an overview of the basic association rules and frequent episodes algorithms. In order to efficiently compute only the "relevant" (i.e., "useful") patterns from the large amount of audit data, we need to utilize domain knowledge in the data mining process. We describe our extensions to these algorithms, which are mechanisms for incorporating the characteristics of audit records.

5.1 The Challenges

In chapter 4 we described in detail our experiments in building classification models to detect intrusions to *sendmail* and TCP/IP networks. The results on the *sendmail* system call data showed that we needed to use as much as 80% of the exhaustively gathered normal data to learn RIPPER classifications rules that can clearly identify normal *sendmail* executions and intrusions. The results on the *tcpdump* of network traffic data showed that by the temporal nature of network activities, when added with temporal and statistical features, the classification model had a very significant improvement in identifying intrusions. These experiments revealed that we need to solve some very challenging problems in order for the classification models to be effective.

Formulating the classification tasks, i.e., determining the class labels and the set of features, from audit data is a very difficult and time-consuming task. Since security is usually an after-thought of computer system design, there is no standard auditing mechanisms and data format specifically for intrusion analysis purposes. A considerable amount of data pre-processing, which involves domain knowledge, is required to extract raw "action" level audit data into higher level "session/event" records with the set of intrinsic system features. The temporal nature of event sequences in network-based computer systems suggests that temporal and statistical measures over the features, e.g., *the average duration of connections in the past 30 seconds*, need to be considered as additional features. Traditional feature selection techniques, as discussed in the machine learning literature, cannot be directly applied here since they don't consider (across record boundary) sequential correlation of features. In [Fawcett and Provost, 1996] Fawcett and Provost presented some very interesting ideas on automatic selection of features for a cellular phone fraud detector. An important assumption in that work is that there are some general patterns of fraudulent usage for the entire customer population, and individual customers differ in the “threshold” of these patterns. Such assumptions do not hold here since different intrusions have different targets on the computer system, and normally produce different evidence and in different audit data sources.

A critical requirement for using classification rules in an anomaly detector is that we need to have “sufficient” training data that covers as much variation of the normal behavior as possible, so that the *false positive* rate is kept low, i.e., we wish to minimize detected “abnormal normal” behavior. It is not always possible to formulate a classification model to learn the anomaly detector with “insufficient” training data, and then incrementally update the classifier using on-line learning algorithms. This is because the limited training data may not have covered all the class labels, and on-line algorithms, for example, ITI [Utgoff *et al.*, 1997], can’t deal with new data with new (i.e., unseen) class labels. For example in modeling daily network traffic, we use the services, e.g., *http*, *telnet* etc., of the connections as the class labels in training models. We may not have connection records for the infrequently used services with, say, only one week’s traffic data. A formal audit data gathering process therefore needs to take place first. As we collect audit data, we need an indicator that can tell us whether the new audit data exhibits any “new” normal behavior, so that we can stop the process when there is no more variation. This indicator should be simple to compute and must be incrementally updated.

5.2 Mining Audit Data

We attempt to develop general rather than intrusion-specific tools in response to the challenges discussed in the previous section. The idea is to first compute the association rules and frequent episodes from audit data, which capture the intra-

and inter- audit record patterns. These frequent patterns can be regarded as the statistical summaries of system activities captured in the audit data, because they measure the correlations among system features and sequential (i.e., temporal) co-occurrences of events. Therefore these patterns can be utilized, with user participation, to guide the audit data gathering and feature selection processes.

In this section we first provide an overview of the basic association rules and frequent episodes algorithms, then describe in detail our extentions that consider the characteristics of audit data.

5.2.1 The Basic Algorithms

From [Agrawal *et al.*, 1993], let $\mathcal{A}$ be a set of attributes, and $\mathcal{I}$ be a set of values on $\mathcal{A}$, called items. Any subset of $\mathcal{I}$ is called an itemset. The number of items in an itemset is called its length. Let $\mathcal{D}$ be a database with n attributes (columns). Define *support*(X) as the percentage of transactions (records) in $\mathcal{D}$ that contain itemset X. An association rule is the expression

$$X \rightarrow Y, [\mathit{confidence}, \mathit{support}]$$

Here X and Y are itemsets, and $X \cap Y = \emptyset$. *support*$(X \cup Y)$ is the support of the rule, and $\frac{\mathit{support}(X \cup Y)}{\mathit{support}(X)}$ is the confidence of the rule. For example, an association rule from the shell command history file (which is a stream of commands and their arguments) of a user is

$$\mathit{trn} \rightarrow \mathit{rec.humor}, [0.3, 0.1]$$

which indicates that 30% of the time when the user invokes *trn*, he or she is reading the news in *rec.humor*, and reading this newsgroup accounts for 10% of the activities recorded in his or her command history file. We implemented the association rules algorithm following the main ideas of Apriori [Agrawal and Srikant, 1994].

```
scan database $\mathcal{D}$ to form $L_1 = \{$frequent 1-itemsets$\}$;
$k = 2$; /* $k$ is the length of the itemsets */
while $L_{k-1} \neq \emptyset$ do begin /* association generation */
        for each pair of $l^1_{k-1}, l^2_{k-1} \in L_{k-1}$ and $l^1_{k-1} \neq l^2_{k-1}$ where
                their first $k-2$ items are the same do begin
            constuct candidate itemset $c_k$ such that its first $k-2$ items are
                the same as $l^1_{k-1}$, and the last two items are the last item of
                $l^1_{k-1}$ and the last item of $l^2_{k-1}$;
            if there is a length $k-1$ subset $s_{k-1} \subset c_k$ and $s_{k-1} \notin L_{k-1}$ then
                remove $c_k$; /* the prune step */
            else
                add $c_k$ to $C_k$;
        end
        scan $\mathcal{D}$ and count the support of each $c_k \in C_k$;
        $L_k = \{c_k | support(c_k) \geq minimum\_support\}$;
        $k = k + 1$;
end
forall $l_k, k > 2$ do begin /* rule generation */
        forall subset $a_m \subset l_k$ do begin
                $conf = support(l_k)/support(a_m)$;
                if $conf \geq minimum\_confidence$ then begin
                    output rule $a_m \rightarrow (l_k - a_m)$,
                            with confidence = $conf$ and support = $support(l_k)$;
                end
        end
end
```

Figure 5.1: Apriori Association Rules Algorithm

Briefly, we call an itemset X a frequent itemset if $support(X) \geq minimum\_support$. Observe that any subset of a frequent itemset must also be a frequent itemset. The algorithm starts with finding the frequent itemsets of length 1, then iteratively computes frequent itemsets of length $k+1$ from those of length k. This process terminates when there are no new frequent itemsets generated. It then proceeds to compute rules that satisfy the *minimum_confidence* requirement. The Apriori algorithm is outlined in Figure 5.1.

Since we look for correlation among values of different attributes, and the (pre-processed) audit data usually has multiple attributes, each with a large number of possible values, we do not convert the data into a binary database as suggested in [Agrawal and Srikant, 1994]. In our implementation we trade memory for speed. The data structure for a frequent itemset has a *row (bit) vector* that records the transactions in which the itemset is contained. The database is scanned only once to generate the list of frequent itemsets of length 1. When a length k candidate itemset c_k is generated by joining two length $k-1$ frequent itemsets l^1_{k-1} and l^2_{k-1}, the row vector of c_k is simply the bitwise AND product of the row vectors of l^1_{k-1} and l^2_{k-1}. The *support* of c_k can be calculated easily by counting the 1s in its row vector, instead of scanning the database. There is also no need to perform the *prune* step in the candidate generation function. The row vectors of length $k-1$ itemsets are freed up to save memory after they are used to generate the length k itemsets. Since most (pre-processed) audit data files are small enough, this implementation works well in our application domain.

The problem of finding frequent episodes based on minimal occurrences was introduced in [Mannila and Toivonen, 1996]. Briefly, given an event database $\mathcal{D}$ where each transaction is associated with a timestamp, an interval $[t_1, t_2]$ is the sequence of transactions that starts from timestamp t_1 and ends at t_2. The width of the interval is defined as $t_2 - t_1$. Given an itemset A in $\mathcal{D}$, an interval is a minimal occurrence of A if it contains A and none of its proper sub-intervals contains A.

Define *support*(X) as the the ratio between the number of minimum occurrences that contain itemset X and the number of records in $\mathcal{D}$. A frequent episode rule is the expression

$$X, Y \rightarrow Z, [\textit{confidence}, \textit{support}, \textit{window}]$$

Here X, Y and Z are itemsets in $\mathcal{D}$. *support*$(X \cup Y \cup Z)$ is the support of the rule, and $\frac{support(X \cup Y \cup Z)}{support(X \cup Y)}$ is the confidence. Here the width of each of the occurrences must be less than *window*. A *serial* episode rule has the additional constraint that X, Y and Z must occur in transactions in partial time order, i.e., Z follows Y and Y follows X. The description here differs from [Mannila and Toivonen, 1996] in that we don't consider a separate *window* constraint on the LHS (left hand side) of the rule. The frequent episode algorithm finds patterns in a single sequence of event stream data. The problem of finding frequent sequential patterns that appear in many different data-sequences was introduced in [Agrawal and Srikant, 1995]. This related algorithm is not used in our study since the frequent network or system activity patterns can only be found in the single audit data stream from the network or the operating system.

Our implementation of the frequent episodes algorithm utilized the data structures and library functions of the association rules algorithm. Here any subset of a frequent itemset must also be a frequent itemset since each interval that contains the itemset also contains all of its subsets. We can therefore also start with finding the frequent episodes of length 2, then length 3, etc. Instead of finding correlations across attributes, we are looking for correlations across records. The *row vector* is now used as the *interval vector* where each pair of adjacent 1s is the pair of boundaries of an interval. A *temporal join* function that considers minimal and non-overlapping occurrences is used to create the interval vector of a candidate length k itemset from the two interval vectors of two length $k-1$ frequent itemsets.

5.2.2 Extensions

These basic algorithms do not consider any domain knowledge and as a result they can generate many "irrelevant" (i.e., uninteresting) rules. In [Klemettinen *et al.*, 1994] rule templates specifying the allowable attribute values are used to post-process the discovered rules. In [Srikant *et al.*, 1997] boolean expressions over the attribute values are used as item constraints during rule discovery. In [Padmanabhan and Tuzhilin, 1998], a "belief-driven" framework is used to discover the "unexpected" (hence "interesting") patterns. A drawback of all these approaches is that one has to know what rules/patterns are interesting or are already in the "belief system". We cannot assume such strong prior knowledge on all audit data.

Interestingness Measures Based on Attributes

We can utilize the schema-level information about audit records to direct the pattern mining process. That is, although we cannot know in advance what patterns, which involve actual attribute *values*, are interesting, we often know what *attributes* are more important or useful given a data analysis task.

Using the minimum support and confidence values to output only the "statistically significant" patterns, the basic algorithms implicitly measure the interestingness (i.e., relevancy) of patterns by their support and confidence values, without regard to any available prior domain knowledge. That is, assume I is the interest-

timestamp	service	src_host	dst_host	src_bytes	dst_bytes	flag	...
1.1	telnet	A	B	100	2000	SF	...
2.0	ftp	C	B	200	300	SF	...
2.3	smtp	B	D	250	300	SF	...
3.4	telnet	A	D	200	12100	SF	...
3.7	smtp	B	C	200	300	SF	...
5.2	http	D	A	200	0	REJ	...
...	...	...	...	...	...	...	...

Table 5.1: Network Connection Records

ingness measure of a pattern p, then

$$I(p) = f(support(p), confidence(p))$$

where f is some ranking function. We propose here to incorporate schema-level information into the interestingness measures. Assume I_A is a measure on whether a pattern p contains the specified important (i.e., "interesting") attributes, our extended interestingness measure is

$$I_e(p) = f_e(I_A(p), f(support(p), confidence(p))) = f_e(I_A(p), I(p))$$

where f_e is a ranking function that first considers the attributes in the pattern, then the support and confidence values.

In the following sections, we describe several schema-level characteristics of audit data, in the forms of "what attributes must be considered", that can be used to guide the mining of relevant features. We do not use these I_A measures in post-processing to filter out irrelevant rules by rank ordering. Rather, for efficiency, we use them as item constraints, i.e., conditions, during candidate itemset generation.

Using the *Axis* Attribute(s)

There is a partial "order of importance" among the attributes of an audit record. Some attributes are essential in describing the data, while others only provide auxiliary information. Consider the audit data of network connections shown in Table 5.1. Here each record (row) describes a network connection. The continuous attribute values, except the timestamps, are discretized into proper bins. A network connection can be uniquely identified by

$$< timestamp, src\_host, src\_port, dst\_host, service >$$

that is, the combination of its start time, source host, source port, destination host, and service (destination port). These are the essential attributes when describing network data. We argue that the "relevant" association rules should describe patterns related to the essential attributes. Patterns that include only the unessential attributes are normally "irrelevant". For example, the basic association rules algorithm may generate rules such as

$$src\_bytes = 200 \rightarrow flag = SF, [0.6, 0.2]$$

These rules are not useful and to some degree are misleading. There is no intuition for the association between the number of bytes from the source, *src_bytes*, and the normal status (*flag* = *SF*) of the connection.

We call the essential attribute(s) *axis* attribute(s) when they are used as a form of item constraint in the association rules algorithm. During candidate generation, an item set must contain value(s) of the axis attribute(s). We consider the correlations among non-axis attributes as not interesting. In other words,

$$I_A(p) = \begin{cases} 1 & \text{if } p \text{ contains axis attribute(s)} \\ 0 & \text{otherwise} \end{cases}$$

In practice, we need not designate all essential attributes as the axis attributes. For example, some network analysis tasks require statistics about various network services while others may require the patterns related to the hosts. We can use *service* as the axis attribute to compute the association rules that describe the patterns related to the services of the connections.

It is even more important to use the axis attribute(s) to constrain the item generation for frequent episodes. The basic algorithm can generate serial episode rules that contain only the "unimportant" attribute values. For example

$$src\_bytes = 200, src\_bytes = 200 \rightarrow dst\_bytes = 300, src\_bytes = 200, [0.4, 0.2, 2s]$$

Note that here each attribute value, e.g., $src\_bytes = 200$, is from a different connection record. To make matter worse, if the support of an association rule on non-axis attributes, $A \rightarrow B$, is high then there will be a large number of "useless" serial episode rules of the form $(A|B)(, A|B)^* \rightarrow (A|B)(, A|B)^*$, due to the following theorem:

Theorem 5.1 *Let s be the support of the association $A \rightarrow B$, and let N be the total number of episode rules on $(A|B)$, i.e., rules of the form*

$$(A|B)(, A|B)^* \rightarrow (A|B)(, A|B)^*$$

then N is at least an exponential factor of s.

Proof:

According to the frequent episodes algorithm, any subset of a frequent itemset is also frequent. For example, if $A, A \rightarrow A, B$ is a frequent episode, then so is $A \rightarrow A$, and $A \rightarrow B$, etc. At each iteration of pattern generation, an itemset is "grown" (i.e., constructed) from its subsets. The number of iterations for growing the frequent itemsets, i.e., the maximum length of an itemset, L, is bounded here by the number of records (instead of the number of attributes as in association rules), which is usually a large number.

At each iteration, we can count N_k, the number of episode patterns on $(A|B)$. Here k is length of such itemsets generated in the current iteration. Thus, the total number of episodes on $(A|B)$ is

$$N = \sum_{k=2}^{L} N_k$$

Assume that there are a total of m records in the database, the time difference between the last and the first record is t seconds. There are $s * m$ records that contain $A \cup B$ in the same transaction. Then the number of minimal and non-overlapping intervals that have k records with $A \cup B$ is $\frac{s*m}{k}$. Notice that each of these intervals therefore contains 2^k length k episodes on $(A|B)$). That is

$$N_k = \frac{s * m}{k} * 2^k$$

and thus

$$N = \sum_{k=2}^{L} \frac{s * m}{k} * 2^k$$

Therefore N is at least an exponential factor of s.

We next show that L monotonically increases with s, but is bounded by m. Assume that the records of the database are evenly distributed with regard to time, and so do the records with $A \cup B$. Then at each iteration, the width of an interval can be as large as $w = \frac{kt}{sm}$. Given the width requirement W, $w \leq W$ must hold, we have

$$k \leq \frac{W * s * m}{t}$$

Therefore L, the maximum value of k, is

$$L = \min\{m, \frac{W * s * m}{t}\}$$

It is easy to see that if the records are not evenly distributed, then w is a factor of $\frac{kt}{sm}$. L still monotonically increases with s. ▌

To avoid having a huge amount of "useless" episode rules, we extended the basic frequent episodes algorithm to compute frequent sequential patterns in two phases: first, it finds the frequent associations using the axis attribute(s); second, it generates the frequent serial patterns from these associations. That is, for the second phase, the items (from which episode itemsets are constructed) are the associations about the axis attribute(s), and the axis attribute values (i.e., length 1 associations). An example of a rule is

$$(\mathit{service} = \mathit{smtp}, \mathit{src\_bytes} = 200, \mathit{dst\_bytes} = 300, \mathit{flag} = \mathit{SF}),$$
$$(\mathit{service} = \mathit{telnet}, \mathit{flag} = \mathit{SF}) \rightarrow$$
$$(\mathit{service} = \mathit{http}, \mathit{src\_bytes} = 200), [0.2, 0.1, 2s]$$

Note that each itemset of the episode rule, e.g.,

$$(\mathit{service} = \mathit{smtp}, \mathit{src\_bytes} = 200, \mathit{dst\_bytes} = 300, \mathit{flag} = \mathit{SF})$$

timestamp	remote host	action	request	...
1	his.moc.kw	GET	/images	...
1.1	his.moc.kw	GET	/images	...
1.3	his.moc.kw	GET	/shuttle/missions/sts-71	...
...	...	...	...	...
3.1	taka10.taka.is.uec.ac.jp	GET	/images	...
3.2	taka10.taka.is.uec.ac.jp	GET	/images	...
3.5	taka10.taka.is.uec.ac.jp	GET	/shuttle/missions/sts-71	...
...	...	...	...	...
8	rjenkin.hip.cam.org	GET	/images	...
8.2	rjenkin.hip.cam.org	GET	/images	...
9	rjenkin.hip.cam.org	GET	/shuttle/missions/sts-71	...
...	...	...	...	...

Table 5.2: Web Log Records

is an association. We in effect have combined the associations among attributes and the sequential patterns among the records into a single rule. This rule formalism not only eliminates irrelevant patterns, it also provides rich and useful information about the audit data.

Using the *Reference* Attribute(s)

Another interesting characteristic of system audit data is that some attributes can be the *references* of other attributes. These reference attributes normally carry information about some "subject", and other attributes describe the "actions" that refer to the same "subject". Consider the log of visits to a Web site, as shown in Table 5.2. Here *action* and *request* are the "actions" taken by the "subject", *remote host*. We see that for a number of remote hosts, each of them makes the same sequence of requests: "/images", "/images" and "/shuttle/missions/sts-71". It is important to use the "subject" as a reference when finding such frequent sequential "action" patterns because the "actions" from different "subjects" are normally irrelevant. This kind of sequential patterns can be represented as

$$\begin{aligned}
&(subject = X, action = a),\\
&(subject = X, action = b) \rightarrow\\
&(subject = X, action = c), [confidence, support, window]
\end{aligned}$$

Note that within each occurrence of the pattern, the *action* values refer to the **same** *subject*, yet the actual *subject* value may not be given in the rule since any particular *subject* value may not be frequent with regard to the entire dataset. In other words, *subject* is simply a *reference* (or a *variable*).

The basic frequent episodes algorithm can be extended to consider reference attribute(s). Briefly, when forming an episode, an additional condition is that, within its minimal occurrences, the records covered by its constituent itemsets have the same value(s) of the reference attribute(s). In other words,

$$I_A(p) = \begin{cases} 1 & \text{if the itemsets of } p \text{ refer to the same reference attribute value} \\ 0 & \text{otherwise} \end{cases}$$

Level-wise **Approximate Mining**

It is often necessary to include the low frequency patterns. In daily network traffic, some services, for example, *gopher*, account for very low occurrences. Yet we still need to include their patterns into the network traffic profile so that we have representative patterns for each supported service. If we use a very low support value for the data mining algorithms, we will then get unnecessarily a very large number of patterns related to the high frequency services, for example, *smtp*.

We use the *level-wise* approximate mining procedure described in Figure 5.2 for finding frequent sequential patterns from audit data. Here the idea is to first

```
Input: database $\mathcal{D}$, the terminating minimum support $s_t$,
       the initial minimum support $s_i$, and the axis attribute(s)
Output: frequent episode rules $Rules$
Begin
(1)     $R_{restricted} = \emptyset$;
(2)     scan database $\mathcal{D}$ to form $L = \{$frequent 1-itemsets that meet $s_t\}$;
(3)     $s = s_i$;
(4)     while $(s \geq s_t)$ do begin
(5)         compute episodes from $L$: each episode must contain at least
                one axis attribute value that is not in $R_{restricted}$;
(6)         append new axis attribute values to $R_{restricted}$;
(7)         append episode rules to the output rule set $Rules$;
(8)         $s = \frac{s}{2}$; /* use a smaller support value for the next iteration */
        end
end
```

Figure 5.2: Level-wise Approximate Mining of Frequent Episodes

find the episodes related to high frequency axis attribute values, for example

$$(service = smtp, src\_bytes = 200),$$
$$(service = smtp, src\_bytes = 200) \rightarrow$$
$$(service = smtp, dst\_bytes = 300), [0.3, 0.3, 2s]$$

We then iteratively lower the support threshold to find the episodes related to the low frequency axis values by *restricting* the participation of the "old" axis values that already have output episodes. More specifically, when an episode is generated, it must contain at least one "new" (low frequency) axis value. For example, in the second iteration, where *smtp* now is an old axis value, we get an episode rule

$$(service = smtp, src\_bytes = 200),$$
$$(service = http, src\_bytes = 200) \rightarrow$$
$$(service = smtp, src\_bytes = 300), [0.4, 0.1, 2s]$$

The procedure terminates when a very low support value is reached. In practice, this can be the lowest frequency of all axis values.

Note that for a high frequency axis value, we in effect omit its very low frequency episodes (generated in the runs with low support values) because they are not as interesting (i.e., representative). In other words, at each iteration we have

$$I_A(p) = \begin{cases} 1 & \text{if } p \text{ contains at least one “new” axis attribute value} \\ 0 & \text{otherwise} \end{cases}$$

Hence our procedure is “approximate” mining. We still include all the old (i.e., high frequency) axis values to form episodes along with the new axis values because it is important to capture the sequential context of the new axis values. For example, although used infrequently, *auth* normally co-occurs with other services such as *smtp* and *login*. It is therefore imperative to include these high frequency services into the episode rules about *auth*.

Our approach here is different from the algorithms in [Han and Fu, 1995] since we do not have (and can not assume) multiple concept levels; rather, we deal with multiple frequency levels of a single concept, e.g., the network service.

Mining with ***Relative Support***

A more flexible approach to deal with attribute values with very skewed distribution is to use *relative support*. Instead of the number of records in the database, the number of occurrences of each unique attribute value in the database can be used as the reference when calculating the support value of a pattern. That is, if a *relative support* value s_i is specified for attribute a_i, and a unique value, e.g., $a_i = v_{ij}$, has a total of n_{ij} occurrences in the database, then a pattern that contains $a_i = v_{ij}$ is frequent if it has at least $s_i * n_{ij}$ occurrences. Different relative support values can be used for different attributes. The procedure of using relative support to mine frequent patterns is outlined in Figure 5.3.

```
**Input**: database $\mathcal{D}$, the overall minimum support $s$,
        and the relative support $s_i$ for each attribute $a_i$
**Output**: frequent patterns (association rules or frequent episodes)
**Begin**
(1)   scan database $\mathcal{D}$ to form $L = \{$unique attribute values$\}$,
          the *count* of each attribute value is recorded;
          *db_size*, the number of records in $\mathcal{D}$, is also recorded;
(2)   **for** each $v_{ij} \in L$ **do begin**
         **if** $s_i$ is specified **then**
            $v_{ij}.rel\_support = s_i * v_{ij}.count$;
         **else**
            **if** $s * v_{ij}.count < db\_size$ **then**
               remove $v_{ij}$ from $L$;
            **else**
               $v_{ij}.rel\_support = s * db\_size$;
      **end** /* $L$ is now the set of frequent 1-itemsets; */
(3)   Compute associations or frequent episodes from $L$:
          when a candidate itemset $l_k$ is constructed from $l^1_{k-1}$ and $l^2_{k-1}$,
          $l_k.rel\_support = \mathbf{min}(l^1_{k-1}.rel\_support, l^2_{k-1}.rel\_support)$,
          $l_k$ is frequent if $l_k.count > l_k.rel\_support$;
**end**
```

Figure 5.3: Mining Frequent Patterns Using Relative Support

Here, step (1) finds all unique attribute values in the database. Step (2) sets the *rel_support* requirements for frequent patterns containing these attribute values, using the *relative support* of an attribute when specified, or the “overall” *support*. Step (3) slightly modifies the candidate generation process for associations or frequent episodes: when a candidate itemset contains values from different attributes that have different relative support requirements, the smallest one is used to check whether the itemset can be a frequent pattern. That is, for each itemset p,

$$I_A(p) = \begin{cases} 1 & \text{if the } \textit{count} \text{ of } p \text{ is no less than the smallest } \textit{relative support} \\ & \text{of its attribute values} \\ 0 & \text{otherwise} \end{cases}$$

This procedure is very useful when we need to compute patterns related to "rare" events, which can be described using several attributes with skewed value distribution. As an example, in network traffic, most connection records have *flag* = *SF* (i.e., normal connection establishment and termination according to the network protocols), we may need to find the cases when the *flag* is not *SF*, e.g., *REJ* (i.e., connection rejected). We can specify (different) relative support requirements for attributes *flag*, *service*, and *dst_host* to uncover the patterns of the "abnormal" traffic. For example, *flag* = *REJ* may occur in only 1% of the connections, so an overall support $s = 0.05$ will filter out any pattern related to *flag* = *REJ*. However, using a relative support for *flag* will very likely produce *flag* = *REJ* patterns since they are measured relative to the number of occurrences of *flag* = *REJ*.

5.3 Summary

In this chapter we first discussed that, in order to construct features to build effective classification models for intrusion detection, we need to mine and analyze the frequent patterns, namely, association rules and frequent episodes, from audit data.

The focus of our research efforts here is on the problem of *how to efficiently compute the "relevant" frequent patterns.* Our solution is to incorporate data schema-level domain knowledge into the data mining process. In effect, we use extended "interestingness" measures on patterns that include information on the attributes, instead of just support and confidence values. These measures are used in our extended algorithms as various forms of item constraints in the mining process. We described in detail these extentions, namely, using *axis* attribute(s) to compute relevant associations, using *reference* attribute(s) to compute valid event episode patterns, using *level-wise* approximate mining to include low frequency but important patterns, and using *relative frequency* to compute the top percent frequent patterns relative to each unique value of the specified attributes.

In Chapter 6, we will discuss how to utilize the mined patterns for audit data gathering and feature construction.

Chapter 6

Managing and Utilizing the Mined Patterns

In this chapter, we describe how to use frequent patterns mined from audit records as guidelines for audit data gathering and feature construction. We first discuss how to incrementally *merge* frequent patterns from multiple audit datasets to form an aggregate rule set, and how to use it as an indicator for audit data gathering. We then present a pattern *encoding* scheme that uses user-defined "order of importance" of audit data attributes. Encoded patterns can then be visualized in 3-D plots for easy understanding and analysis. We describe experiments in visualizing and identifying "intrusion-only" patterns for a number of well-known attacks. We need to automate the process of identifying intrusion patterns and constructing predictive features. We present a procedure for comparing two sets of patterns based on their encodings, and an algorithm for constructing temporal and statistical features from a selected pattern.

6.1 Aggregating the Mined Patterns

We posit that the patterns discovered from the audit data on a protected target (e.g., a network, system program, or user, etc.) corresponds to the target's behavior. When we gather audit data about the target, we compute the patterns from each new audit data set, and *merge* the new rules into the existing aggregate rule set. The added new rules represent (new) variations of the normal behavior. When the aggregate rule set stabilizes, i.e., no new rules from the new audit data can be added, we can stop the data gathering since the aggregate audit data set has covered sufficient variations of the normal behavior.

Our approach of merging rules is based on the fact that even the same type of behavior will have slight differences across audit data sets. Therefore we should not expect perfect (exact) match of the mined patterns. Instead we need to combine similar patterns into more generalized ones.

We merge two rules, r_1 and r_2, into one rule r if

- their right and left hand sides are exactly the same, or their RHSs can be *combined* and LHSs can also be *combined*; and
- the *support* values and the *confidence* values are *close*, i.e., within an ε (a user-defined threshold).

The concept of *combining* here is similar to *clustering* in [Lent *et al.*, 1997] in that we also combine rules that are "similar" syntactically with regard to their attributes, and are "adjacent" in terms of their attribute values. That is, two LHSs (or RHSs) can be combined, if

- they have the same number of itemsets; and

- each pair of corresponding itemsets (according to their positions in the patterns) have the same axis attribute value(s) and adjacent non-axis attributes.

As an example, consider combining the LHSs and assume that the LHS of r_1 has just one itemset,

$$(ax_1 = vx_1, a_1 = v_1)$$

Here ax_1 is an axis attribute. The LHS of r_2 must also have only one itemset,

$$(ax_2 = vx_2, a_2 = v_2)$$

Further, $ax_1 = ax_2$, $vx_1 = vx_2$, and $a_1 = a_2$ must hold. For the LHSs to be combined, v_1 and v_2 must be the same value or adjacent bins of values. The LHS of the merged rule r is

$$(ax_1 = vx_1, v_1 \leq a_1 \leq v_2)$$

(assuming that v_2 is the larger value). For example,

$$(\mathit{service} = \mathit{smtp}, \mathit{src\_bytes} = 200)$$

and

$$(\mathit{service} = \mathit{smtp}, \mathit{src\_bytes} = 300)$$

can be combined into

$$(\mathit{service} = \mathit{smtp}, 200 \leq \mathit{src\_bytes} \leq 300)$$

To compute the statistically relevant support and confidence values of the merged rule r, we record *support_lhs* and *db_size* of r_1 and r_2 when mining the rules from the audit data. Here *support_lhs* is the support of a LHS and *db_size* is the number of records in the audit data. The support value of the merged rule r is

$$\mathit{support}(r) = \frac{\mathit{support}(r_1) * \mathit{db\_size}(r_1) + \mathit{support}(r_2) * \mathit{db\_size}(r_2)}{\mathit{db\_size}(r_1) + \mathit{db\_size}(r_2)}$$

and the support value of LHS or r is

$$\mathit{support\_lhs}(r) = \frac{\mathit{support\_lhs}(r_1) * \mathit{db\_size}(r_1) + \mathit{support\_lhs}(r_2) * \mathit{db\_size}(r_2)}{\mathit{db\_size}(r_1) + \mathit{db\_size}(r_2)}$$

and therefore the confidence value of r is

$$\mathit{confidence}(r) = \frac{\mathit{support}(r)}{\mathit{support\_lhs}(r)}$$

6.1.1 Experiments on Audit Data Gathering

We now describe experiments that tested our hypothesis that the merged rule set can indicate whether the audit data has covered sufficient variations of behavior. We obtained one month of TCP/IP network traffic data from `http://ita.ee.lbl.gov/html/contrib/LBL-CONN-7.html`. We hereafter refer it as the LBL dataset. There are total about 780,000 connection records. We segmented the data by day. And for data of each day, we again segmented the data into four partitions: morning, afternoon, evening and night. This partitioning scheme allowed us to cross evaluate anomaly detection models of different time segments that have different traffic patterns. It is often the case that very little (and sometimes no) intrusion data is available when building an anomaly detector. A common practice is to use audit data of legitimate activities that is known to have different behavior patterns for testing and evaluation.

Here we describe the experiments and results on building anomaly detection models for the "weekday morning" traffic data on connections originated from LBL

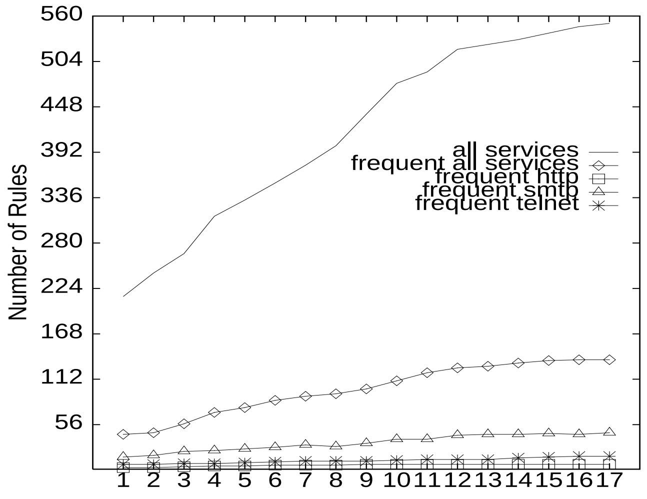


Figure 6.1: The Number of Rules vs. the number of Audit Data Sets

to the outside world. There are about 137,000 such connections for the month. We decided to compute the frequent episodes using the network *service* as the axis attribute. Recall from our earlier discussion that this formalism captures both association and sequential patterns. For the first three weeks, we mined the patterns from the audit data of each weekday morning, and merged them into the aggregate rule set. For each rule we recorded *merge_count*, the number of merges on this rule. Note that if two rules r_1 and r_2 are merged into r, its *merge_count* is the sum from the two rules. *merge_count* indicates how frequent the behavior represented by the merged rule is encountered across a period of time (days). We call the rules with *merge_count* $\geq$ *min_frequency* the frequent rules.

Figure 6.1 plots how the rule set changes as we merge patterns from each new audit data set. We see that the total number of rules keeps increasing. We visually inspected the new rules from each new data set. In the first two weeks, the majority are related to "new" network services that have no prior patterns in the aggregate rule set. And for the last week, the majority are just new rules of the existing services. Figure 6.1 shows that the rate of change slows down during the last week. We used *min_frequency* $= 2$ to filter out the "one-time" patterns and called the remaining ones the "frequent rules". We can see in the figure that, the frequent rule sets of all services as well as the individual services grow at much slower rates and tend to stabilize.

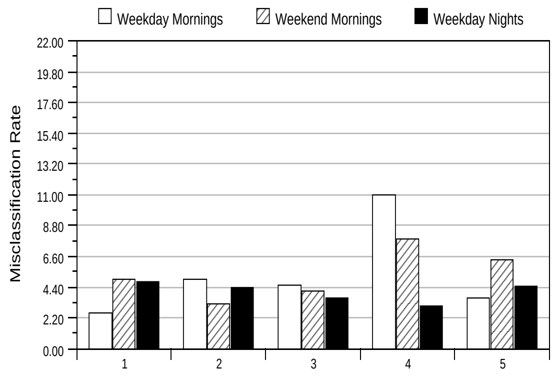


Figure 6.2: Misclassification Rates of Classifier Trained on First 8 Weekdays

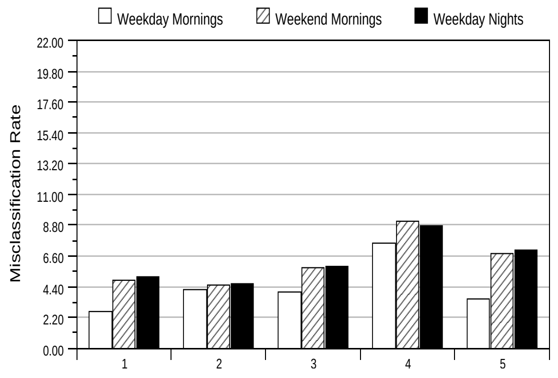


Figure 6.3: Misclassification Rates of Classifier Trained on First 10 Weekdays

We used the set of frequent rules of all services as the indicator on whether the audit data is sufficient. We tested the quality of this indicator by constructing four classifiers, using audit data from the first 8, 10, 15, and 17 weekday mornings, respectively, for training. We used the services of the connections as the class labels, and included a number of temporal statistical features, for example, the average of *duration* of the connections in the past 140 seconds, the average of *data bytes* from source to destination of the connection in the past 140 seconds, etc.. The classifiers were tested using the audit data that was not used in training, and was from the mornings and nights of the last 5 weekdays of the month, as well as the last 5 weekend mornings. Figures 6.2, 6.3, 6.4, and 6.5 show the performance of these four classifiers in detecting anomalies (i.e., different behavior) respectively. In each figure, we show the misclassification rate (i.e., percentage of misclassifications) on the test data. Since the classifiers model the weekday morning traffic, we wish to see this rate to be low on the weekday morning test data, but high on the weekend morning data as well as the weekday night data. The figures show that the classifiers with more training audit data perform better. Further, the last two classifiers are effective in detecting anomalies, and their performance are very close (see figures 6.4 and 6.5). This is not surprising at all because from the plots in figure 6.1, the set of frequent rules (i.e., our indicator on audit data) is growing in weekday 8 and 10, but stabilizes from day 15 to 17. Thus this indicator on audit data gathering is quite reliable.

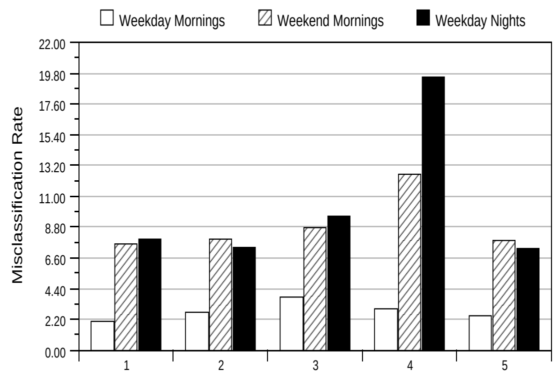


Figure 6.4: Misclassification Rates of Classifier Trained on First 15 Weekdays

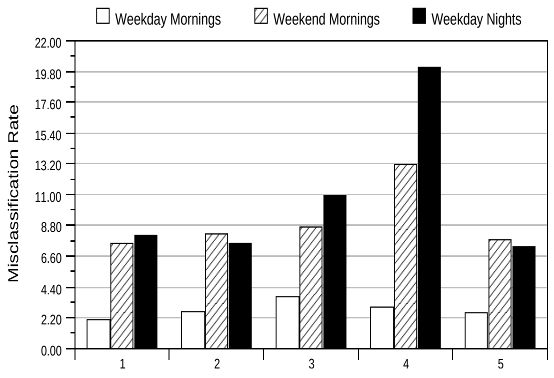


Figure 6.5: Misclassification Rates of Classifier Trained on First 17 Weekdays

6.1.2 Off-line Analysis

Since the merged rule set was used to identify and collect "new" behavior during the audit data gathering process, one naturally asks "Can the final rule set be directly used to detect anomalies?". Here we used the set of frequent rules to distinguish the traffic data of the last 5 weekday mornings from the last 5 weekend mornings, and the last 5 weekday nights. We use a *similarity* measure. Assume that the merged rule set has n rules, and the size of the new rule set from a new audit data set is

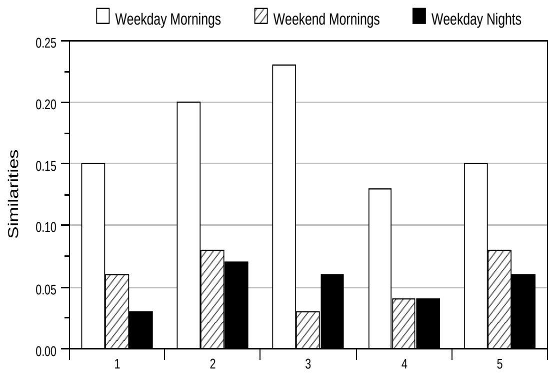


Figure 6.6: Similarity Measures Against the Merged Rule Set of Weekday Mornings

m, the number of *matches* (i.e., the number of rules that can be merged) between the merged rule set and the new rule set is p, then we have

$$similarity = \frac{p}{n} * \frac{p}{m}$$

Here $\frac{p}{n}$ represents the percentage of known behavior (from the merged rule set) exhibited in the new audit data, and $\frac{p}{m}$ represents the proportion of (all) behavior in the new audit data that conforms to the known behavior. Figure 6.6 shows that the *similarity* of the weekday mornings are much larger than the weekend mornings and the weekday nights.

In general the mined patterns cannot be used directly to classify the records (i.e., they cannot tell *which* records are anomalous). They are very valuable in off-line analysis. By studying the differences between frequent pattern sets, we can identify the different behavior across audit data sets. For example, by comparing the patterns from normal and intrusion data, we can gain a better understanding of the nature of the intrusions and identify their "signature" patterns.

6.2 Visualizing and Analyzing the Mined Patterns

The purpose of data mining is to discover potentially useful knowledge from the data. End-users often need to inspect the mined patterns to select the "interesting" ones for further study or evaluation. Supporting the presentation, analysis and comparison of the mined patterns should therefore be considered as an integral part of the knowledge discovery process.

Simply ranking the discovered patterns by their statistical significance (e.g., the confidence and support values) is not necessary the right approach since the most frequent patterns may represent "known" knowledge, or worse, may just be artifacts in the data (i.e., they bear no useful knowledge). In [Klemettinen *et al.*, 1994] rule templates specifying the allowable attribute values are used to select only a small number of patterns. The main drawback of this approach is that it will miss some truly "interesting" and "surprising" patterns that do not fit the rule templates because the end-users might not have known them a priori.

The frequent sequential patterns generated by our algorithms have some interesting characteristics:

- Not all attribute values presented in a pattern provide useful information. For example, in

  $$(service = icmp\_echo, src\_host = host_A, duration = 0),$$
  $$(service = icmp\_echo, src\_bytes = 0, dst\_bytes = 0, hole\_rate = 0) \rightarrow$$
  $$(service = icmp\_echo, src\_host = host_A, src\_bytes = 0), [0.5, 0.1, 5s]$$

  attributes *duration*, *hole_rate*, *src_bytes* and *dst_bytes* are meaningless since for the *icmp_echo* service, these attributes always have value 0. In fact, they are not essential in describing network connections in general. The end-users may thus want to first see a pattern only in terms of the important attributes, and if necessary, examine it with other attributes as well. More generally, it is appropriate to present a pattern in multiple phases: first with only the most important attribute, then with the second most important attribute also included, etc. In other words, we need a general-to-specific presentation of the patterns.

- There are natural "clusters" of patterns, especially when they are presented with only the important attributes. That is, several patterns can carry "similar" (high-level) information about the data. For example, consider the patterns

  $$\begin{aligned}&(\mathit{service} = \mathit{icmp\_echo}, \mathit{src\_bytes} = 0),\\ &(\mathit{service} = \mathit{icmp\_echo}, \mathit{dst\_bytes} = 0) \rightarrow\\ &(\mathit{service} = \mathit{icmp\_echo}, \mathit{src\_bytes} = 0), [0.5, 0.1, 5s]\end{aligned}$$

  and

  $$\begin{aligned}&(\mathit{service} = \mathit{icmp\_echo}, \mathit{dst\_bytes} = 0),\\ &(\mathit{service} = \mathit{icmp\_echo}, \mathit{src\_host} = \mathit{host}_A, \mathit{flag} = \mathit{SF}) \rightarrow\\ &(\mathit{service} = \mathit{icmp\_echo}, \mathit{src\_bytes} = 0), [0.5, 0.1, 5s]\end{aligned}$$

  When using only *service*, the axis attribute here, they can both be summarized as "*there are frequent sequences of* icmp_echo *connections*". It is desirable to provide such a summary for each cluster so that end-users can quickly determine which one deserves further analysis. And the individual patterns of the selected cluster can then be examined.

Based on these observations, we developed a visualization scheme that can show all the mined patterns in clusters, and can support general-to-specific analysis.

6.2.1 Representing the Mined Patterns

In order to show clusters of patterns, we need to first encode each pattern in such a way that "similar" patterns are mapped to data points that are close to each other. Many clever techniques, for example [Faloutsos and Lin, 1995], have been proposed to map objects into points in k-dimensional space so that distances (i.e., similarities) are preserved as well as possible. These approaches are useful in cluster analysis of data. However we are dealing with a very different problem: the visualization and analysis of patterns (rules). It is imperative that we present the "structural content", i.e., the LHSs and RHSs, of the rules. That is, it is not sufficient to simply show that there are several clusters of patterns, it is more important to display the information represented by the patterns in a cluster.

Preserving the Rule Structure

We first consider the structure of the frequent sequential patterns. For a serial rule,

$$X, Y \rightarrow Z, [\mathit{confidence}, \mathit{support}, \mathit{window}]$$

where X, Y, and Z are associations (itemsets), we can interpret it as: if a record with pattern X is followed by another record with pattern Y, c percent of the time, there is yet another following record with pattern Z, and X, Y and Z are within the same time *window*. If we can encode each association as a number, we can then map a sequential pattern into a 3-d data point $(\mathit{encoding}_X, \mathit{encoding}_Y, \mathit{encoding}_Z)$. That is, intuitively the z axis represents the last record, and the x and y axes represent the first two records.

A limitation of this representation scheme, due to difficulties in manipulating general n-dimensional (where $n > 3$) displays, is that we can only encode at most two associations in the LHS and one association in the RHS. For an arbitrary rule

$$L_1, L_2, ..., L_i \rightarrow R_1, R_2, ..., R_j$$

where $i, j \geq 1$, we take its general (subsuming) form

$$L_1, L_2 \rightarrow R_j$$

and compute the encoding. As a result, rules having the same L_1, L_2, and R_j are mapped to the same data point in a 3-d space. If $i = 1$, we simply assign $encoding_Y = 0$. The *support*, *confidence*, and *window* values are not represented in the 3-d display. The original rule(s), in its entirety, can be presented to the end-users if the corresponding data point is selected.

Encoding the Associations

The goal of our encoding scheme is to map associations, and thus the sequential patterns, that are structurally and syntactically more "similar" to closer numbers. We also seek an encoding scheme that is simple to compute and manipulate. To define the "similarity" measure precisely, we first define a partial order on all the discovered associations. Assuming n-dimensional data, we call an association

$$(A_1 = v_1, A_2 = v_2, ..., A_k = v_k)$$

"complete and ordered" if $k = n$ and attributes $A_1, A_2, ..., A_k$ are in the order of decreasing importance. For example, A_1 is the axis attribute, $A_2, ..., A_i$ $(i \leq n)$ are other important attributes (in decreasing order), and $A_{i+1}, ..., A_n$ are the other attributes in some alphabetical order. Or in the simplest form, the attributes can just be ordered alphabetically. A discovered association can always be converted to its "complete and ordered" form by first inserting $A_i = null$ for each "missing" attribute A_i, and then sorting the attributes in the order of importance. For two

```
Global variables:
   values[$n$][]; /* assuming $n$-dimensional data, values[$j$] holds the
                    unique values of attribute $A_j$ */
   count[$n$]; /* count[$j$] is the number of unique values of $A_j$ */
   codes[]; /* code[$i$] is the (final) encoding of association $X_i$ */
   tempCodes[][]; /* tempCodes[$i$][$j$] stores the encoding for attribute
                      value $v_j$ in $X_i$ */
begin
(1)      for each association $X_i$
              convert $X_i$ to "complete and order" form;
              for each attribute value $v_j$ in $X_i$
                   insert $v_j$ in values[$j$] in sorted order, and
                           increment count[$j$] if needed;

(2)      for each association $X_i$
              for each attribute value $v_j$ in $X_i$
                   if $v_j$ is null /* attribute is "missing" */
                      tempCodes[$i$][$j$] = 0;
                   else
                      tempCodes[$i$][$j$] = $k$, for values[$j$][$k$] = $v_j$;

(3)      let maxCount = the largest in count[], and $a = \log_{10} maxCount$;

(4)      if $a \geq 2$
            /* the values of at least one attributes can not be stored
               in single digit */
            for each $b = \log_{10}$ count[$j$] where $b < a$
                 /* for attributes whose values can be stored in less than
                    $a$ digits, update the encoding of each $X_i$, so that
                    all attributes use $a$ digits to store the values */
                 add $10^{a-1}$ to each tempCodes[$i$][$j$];

(5)      for each $X_i$
              codes[$i$] = tempCodes[$i$][1]$\times(10^a)^{n-1}$ + tempCodes[$i$][2]$\times(10^a)^{n-2}$
                           + ... + tempCodes[$i$][$n-1$]$\times 10^a$ + tempCodes[$i$][$n$];
end
```

Figure 6.7: Algorithm *Encoding*

"complete and ordered" associations, we say

$$(A_1 = v_1, A_2 = v_2, ..., A_n = v_n) < (A_1 = u_1, A_2 = u_2, ..., A_n = u_n)$$

if $v_j = u_j$, for $j = 1, 2, ..., i-1$, and $v_i < u_i$. It is assumed here that the values of an attribute, e.g., v_i and u_i, are compared numerically for a continuous attribute, or alphanumerically for a discrete attribute. We say association X_i is more "similar" to X_j than to X_k if $X_i < X_j < X_k$ (or $X_k < X_j < X_i$) holds.

Figure 6.7 outlines the encoding procedure that preserves the "similarity" measures of associations. Here step (1) sorts the unique values of each attribute. Step (2) assigns a temporary encoding to each association using the sorted orders of its attribute values. Steps (3) and (4) ensure that each attribute "occupies" the same number of digits so that when comparing the encodings of two associations, the digits for the corresponding attributes are aligned properly. Step (5) computes the final encoding of an association in such a way that the order of significance in the digits corresponds to the order of importance of the attributes. It can be easily proved that this encoding algorithm is correct, i.e., if X_i is more "similar" to X_j than to X_k, then

$$|Encoding_{X_i} - Encoding_{X_j}| < |Encoding_{X_i} - Encoding_{X_k}|$$

Assume that the number of associations is N, the worst-case running time of Algorithm *Encoding* is $O(N \log N)$ because in the sorting operations in step (1), count[j] can be proportional to N. However, the number of unique "frequent" values of any attribute (i.e., count[j] for $j = 1, ..., n$) is normally a small constant due to the *support* threshold, therefore the average-case (i.e., expected) running time of *Encoding* is $O(N)$.

association	encoding
$(service = http, src\_host = host_A, flag = SF)$	1101
$(service = icmp\_echo, dst\_host = host_B)$	2010
$(service = http, flag = REJ)$	1002
$(service = unknown, src\_host = host_A)$	3100
$(service = icmp\_echo, src\_host = host_C, dst\_host = host_B, flag = REJ)$	2212
...	...

Table 6.1: Encodings of Associations for Visualization

Table 6.1 shows some examples of encodings. The order of importance among the attributes is *service*, *src_host*, *dst_host*, and *flag*. We can see that associations that are more "similar" (i.e., they share the same values on their important attributes) are mapped to closer numbers.

An advantage of our encoding scheme is that we can use simple arithmetic operations to very easily control the "level of detail" required for analysis or comparison of the associations. For example, if we choose to "ignore" *flag*, we can simply do an integer division by 10 on the encodings.

6.2.2 Analysis of Patterns

A number of pattern analysis tasks can be readily supported by simple manipulations on the encodings. Here *drill-down* and *roll-up* have similar spirits as their counterparts in OLAP, but are much simpler operations since they deal with encoded patterns.

- *Drill-down*: examine the patterns in terms of the less important attributes. For example, after identifying a cluster based on the values on the most important attribute, the user may want to look into its patterns to study their differences. This can be accomplished by using the arithmetic modulus operation to throw away the most significant digit, and updating the display using the new encodings, where now the most significant digit represents the second most important attribute.

- *Roll-up*: view the patterns in terms of only the more important attributes. For example, the user may want to reduce each cluster to a representative point in order to produce a summary report. This can be accomplished by using integer division to truncate the less significant digits of the encodings.

- *Segmentation*: find patterns that match a rule template, which specifies the allowable attribute values for each of the associations in a rule. The rule template is first encoded. Then the data points that match the encoded template in the corresponding digits are selected for display.

6.2.3 Experiments on IWSS16 Dataset

We describe here experiments on a set of network intrusion data from InfoWorld. This dataset contains attacks of the "InfoWorld Security Suite 16" (IWSS16) that was used to evaluate several leading commercial intrusion detection products [McClure *et al.*, 1998]. We were given two traces of *tcpdump* data. One contains 4 hours of normal network traffic, and the other contains 2 hours of network traffic where 16 different types of attacks were simulated. We first summarized the raw (packet level) *tcpdump* data into connection records using the procedures described in Section 4.3.2.

According to their attack methods[1], several intrusions in IWSS16 would leave

[1]Scripts and descriptions of many intrusions can be found using the search engine in `http:`

distinct evidence in the short sequence of time ordered connection records. Below we select a few of these intrusions as examples to demonstrate how our data mining algorithms can be used to find the patterns of these attacks. We also illustrate the utility of our visualization techniques in comparing and analyzing the patterns from the normal and intrusion data. Here we use a time window of 2 seconds. And the *level-wise* mining procedure is employed.

Ping Scan

The attacker systematically sends ping (i.e., *icmp echo*) requests to a large number of different hosts to find out which host is available. In the connection records, there should be a host that makes *icmp echo* connections to many different hosts in a short period of time.

- Data mining strategy: use *service* as the axis attribute and *src_host* as the reference attribute to find the "same source host" frequent sequential "service" patterns;

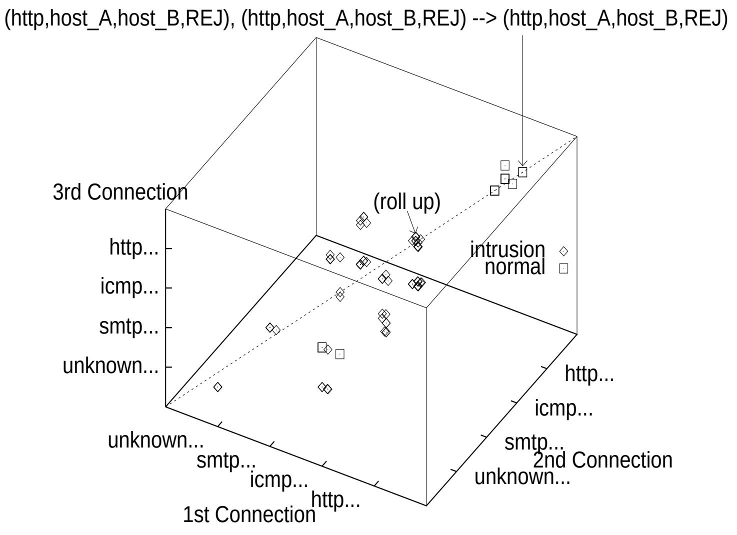
- Evidence in intrusion data: there are several patterns that suggest the attack, for example,

$$(service = icmp\_echo, src\_host = host_h),$$
$$(service = icmp\_echo, src\_host = host_h) \rightarrow$$
$$(service = icmp\_echo, src\_host = host_h), [0.4, 0.1, 2s]$$

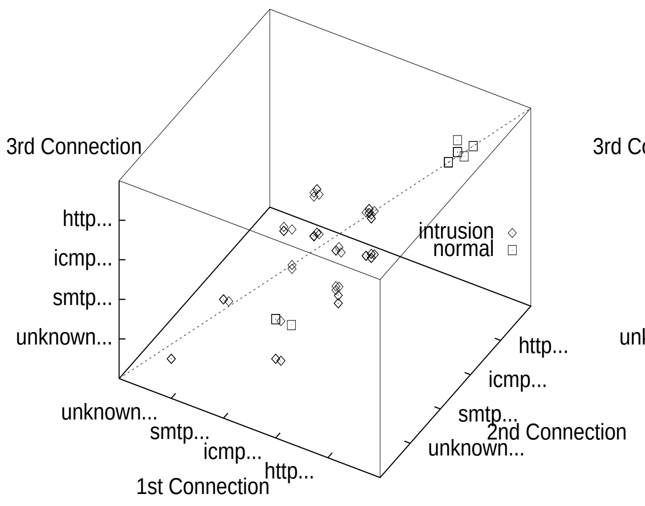
  Note that there is no *dst_host* in this rule, suggesting that *icmp echo* is sent to "different" hosts;

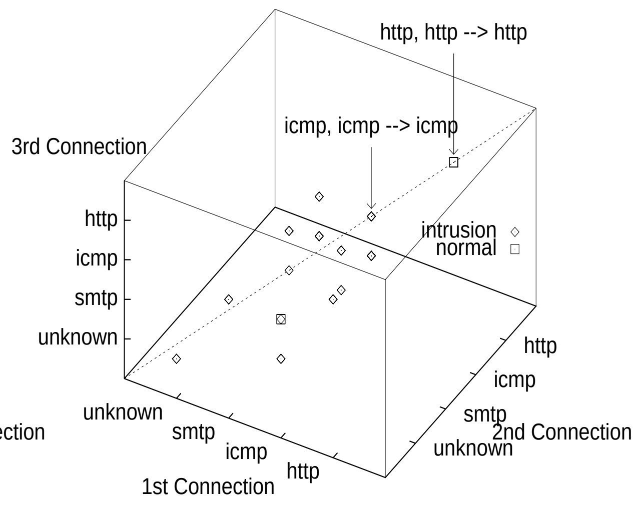
- Contrast with normal data: no such patterns.

`//www.rootshell.com`.



(a) Patterns





(b) Roll-up 2 attributes

(c) Roll-up 3 attributes

Figure 6.8: Ping Scan

- Visualization: We encode the patterns from the normal and intrusion data in the same pass (i.e., by supplying them to the *Encoding* program together) so that both sets of patterns can be displayed in the same 3-d space. Figure 6.8 (a) shows the 3-d plots of the mined patterns. Based on the data mining strategy, we use this "order of importance" for encoding: *service*, *src_host*, *dst_host* and *flag* (we omit other attributes). Each axis is labeled with the values of the most significant attribute instead of using their numerical encodings. For example, "http ..." represents the encoding of associations of the form

  $$(\mathit{service} = \mathit{http}, \mathit{src\_host} = *, \mathit{dst\_host} = *, \mathit{flag} = *)$$

  Along the diagonal line, a data point represents the pattern of the form $A, A \rightarrow A$ (i.e., frequent sequence of the same "type" of connections). We can see that patterns indeed tend to form clusters. Further, the patterns from the normal and intrusion datasets form different and distant clusters, and only the intrusion data has patterns of repeated *icmp_echo* connections. Figures 6.8 (b) and (c) are examples of the roll-up analysis. As we discard 2 and 3 less important attributes, the clusters are reduced to summary data points.

Syn Flood

The attacker makes a lot of "half-open" connections by sending only a "syn request" but not establishing the connection to a port of a target host in order to fill up the victim's connection-request buffer and achieve "denial-of-service". In the connection records, there should exist a host where one of its port receives a lot of connections with flag "S0" (i.e., only the "syn request" packet is seen) in a short period of time.

- Data mining strategy: use *service* as the axis attribute and *dst_host* as the reference attribute to find the "same destination host" frequent sequential "service" patterns;

- Evidence in intrusion data: there is very strong evidence of the attack, for example,

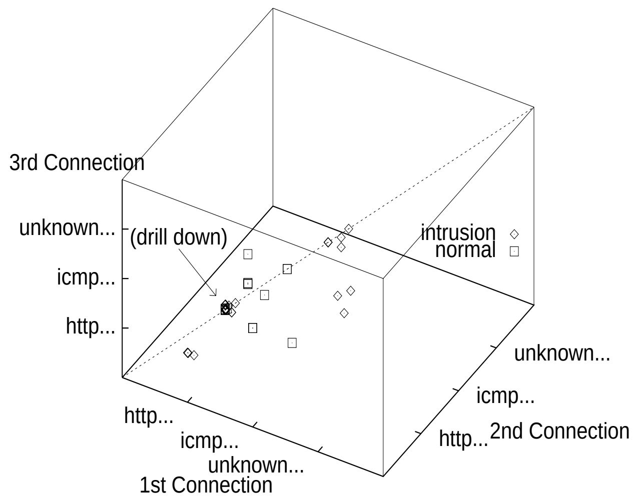
$$\begin{aligned}&(\mathit{service} = \mathit{http}, \mathit{flag} = \mathit{S0}),\\ &(\mathit{service} = \mathit{http}, \mathit{flag} = \mathit{S0}) \rightarrow\\ &(\mathit{service} = \mathit{http}, \mathit{flag} = \mathit{S0}), [0.6, 0.15, 2s]\end{aligned}$$

- Contrast with normal data: no patterns with $\mathit{flag} = \mathit{S0}$.

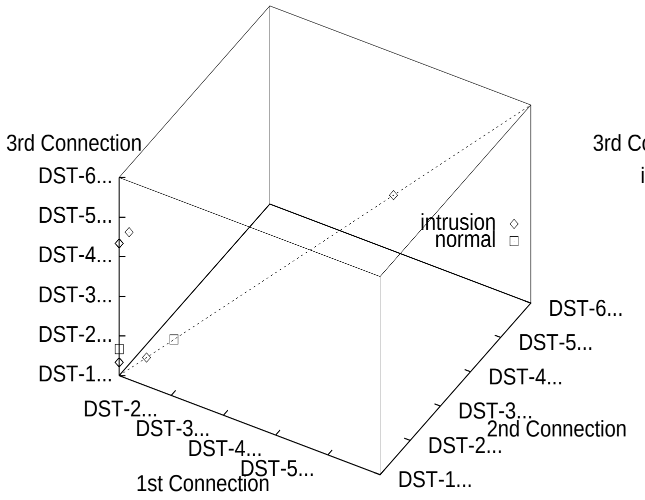
- Visualization: Figure 6.9 (a) shows the 3-d plot of the mined patterns. Here the order of importance is: *service*, *dst_host*, *flag*, and *src_host*. From (a) we see that although there are distinct clusters for patterns from the intrusion and normal datasets, there is also an area (as identified by the arrow) of "overlap". Figure 6.9 (b) shows the result of drill-down analysis. We can see that the patterns from the two datasets refer to different *dst_host*, i.e., they are not really "overlapped". Figure 6.9 (c) shows the result of segmentation analysis using the template

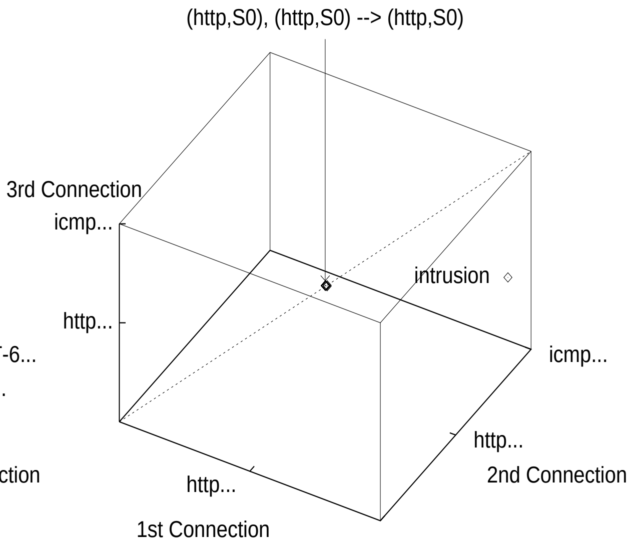
$$(\mathit{flag} = \mathit{S0}), (\mathit{flag} = \mathit{S0}) \rightarrow (\mathit{flag} = \mathit{S0})$$

  We see that only the intrusion data has such patterns, and the victim (target) port is indeed *http*.



(a) Patterns





(b) Drill-down

(c) Segmentation

Figure 6.9: Syn Flood

Port Scan and Netsonar Scan

The attacker systematically makes connections to each port (service) of a target host (i.e., the destination host) in order to find out which ports are accessible. In the connection records, there should be a victim host (or hosts) that receives many connections to its "different" ports in a short period of time. Further, a lot of these connections have the "REJ" flag since many ports are normally unavailable and hence the connections are rejected.

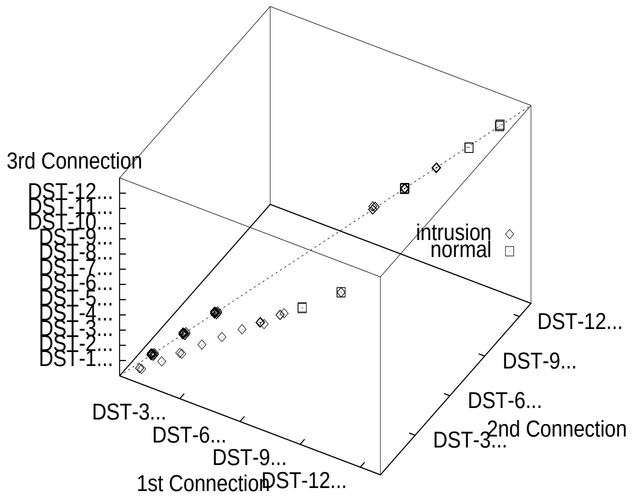
- Data mining strategy: use *dst_host* as both the axis attribute and the reference attribute to find the "same destination host" frequent sequential "destination host" patterns;

- Evidence in intrusion data: there are several patterns that suggest the attack, for example,

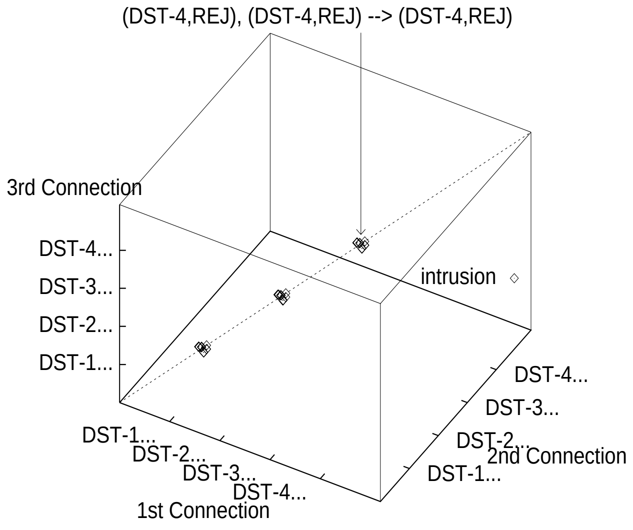
  $$\begin{aligned}&(\mathit{dst\_host} = \mathit{host}_v, \mathit{flag} = \mathit{REJ}),\\&(\mathit{dst\_host} = \mathit{host}_v, \mathit{flag} = \mathit{REJ}) \rightarrow\\&(\mathit{dst\_host} = \mathit{host}_v, \mathit{flag} = \mathit{REJ}), [0.65, 0.2, 2s]\end{aligned}$$

  but no patterns with $\mathit{flag} = \mathit{REJ}$ are found when using the *service* as the axis attribute (and *dst_host* as the reference attribute) since a large number of difference services (ports) are attempted in a short period time. As a result, for each service the "same destination host" sequential patterns are not frequent;

- Contrast with normal data: patterns related to $\mathit{flag} = \mathit{REJ}$ indicate that the "same" service is involved.



(a) Patterns



(b) Segmentation

Figure 6.10: Port Scan

- Visualization: Figure 6.10 (a) shows the 3-d plot of the mined patterns. Here the order of importance is: *dst_host*, *flag*, *src_host*, and *service*. We see the patterns form clusters along the diagonal line of the cube, as well as the diagonal line of the plane of "1st connection" and "3rd connection". This simply says that these hosts get connected frequent enough during the short time window. For example, the patterns in the diagonal line have the form

  $$(\mathit{dst\_host} = \mathit{host}_x, ...), (\mathit{dst\_host} = \mathit{host}_x, ...) \rightarrow (\mathit{dst\_host} = \mathit{host}_x, ...)$$

  Figure 6.10 (b) shows the result of segmentation using the template

  $$(\mathit{flag} = \mathit{REJ}), (\mathit{flag} = \mathit{REJ}) \rightarrow (\mathit{flag} = \mathit{REJ})$$

  We see that there are three hosts that fall victims to the port scan attacks.

6.3 Feature Construction from Mined Patterns

When packet data is summarized into the connection records (see Figure 3.1) using commonly available packet processing engines, each record contains a set of "intrinsic" features that are for general network traffic analysis purposes. These features are: *service*, *src_host*, *src_port*, *dst_host*, *dur* (duration of the connection), and *src_bytes* and *dst_bytes* (number of data bytes from each direction). The frequent sequential patterns from these initial connection records can be viewed as statistical summaries of the network activities. In Section 6.2.3, we showed that by comparing the patterns from a "normal" dataset (e.g., data collected from normal network traffic over an extended period of time) and an "intrusion" dataset (e.g., data recorded from simulation runs of attack programs), we can identify the "intrusion-only" patterns for several representative network attacks. Since our ultimate goal is to produce effective classifiers to detect these intrusions, we are interested in how to utilize these intrusion patterns to construct predictive features into the classification models.

The choice of the axis and reference attributes is very important in computing and identifying the intrusion-only patterns. For example, using *dst_host* as both the axis and reference attribute produces very distinct "port scan" patterns, e.g.,

$$\begin{aligned}
&(\mathit{dst\_host} = \mathit{host}_v, \mathit{flag} = \mathit{REJ}),\\
&(\mathit{dst\_host} = \mathit{host}_v, \mathit{flag} = \mathit{REJ}) \rightarrow\\
&(\mathit{dst\_host} = \mathit{host}_v, \mathit{flag} = \mathit{REJ}), [0.65, 0.2, 2s]
\end{aligned}$$

But no intrusion pattern is found when using the *service* as the axis attribute and *dst_host* as the reference attribute since a large number of different services are attempted in a short period time, and as a result, for each service the "same destination host connection rejected" sequential patterns are not frequent.

We need to alleviate the user from the burden of "guessing" these choices. An iterative procedure that involves pattern mining and comparison, feature construction from patterns, and model building and evaluation, is thus employed. In each iteration, a different combination of axis attribute and reference attribute is selected. The choices are limited among the essential attributes, that is, *service*, *dst_host*, *src_dst*, or *src_port*. Note that the exact *time* is never frequent and is thus omitted. Since intrusions are generally targeted to some victim host(s) in the network, we start with *service* as the axis attribute and *dst_host* as the reference attribute. For each iteration, the set of features along with the performance of the resulting classifier, in both TP (true positive) and FP (false positive) is recorded.

The set of features that results in the best model is selected at the end of this procedure. We next discuss pattern comparison and feature construction here.

6.3.1 Pattern Comparison

The goal of the *pattern comparison* process is to identify the intrusion-only patterns. The main tasks here include

- Create the "baseline" normal pattern sets by mining the frequent patterns from each subset (e.g., each day) of normal audit records, and incrementally adding the patterns to an aggregate pattern set. This is done for each possible combination of axis and reference attributes. That is, each combination results in a pattern set.
- Mine frequent patterns from intrusion audit records.
- Compare the patterns from the intrusion data with the corresponding normal pattern set. That is, the two sets of patterns here have the same choice of axis and reference attributes.

We focus our discussion on how to efficiently compare two sets of patterns. The aggregate normal pattern set is usually very large, in the range of several thousands to tens of thousands of patterns. We therefore use the encoding scheme discussed in Section 6.2.1 to convert each frequent pattern to a number so that we can easily compare two sets of patterns.

Table 6.2 shows some examples of encodings. Here *service* is the *axis* attribute, and *dst_host* is the reference attribute. When encoding associations from

association	encoding
$(\mathit{flag} = \mathit{SF}, \mathit{service} = \mathit{http}, \mathit{src\_bytes} = 200)$	11001
$(\mathit{service} = \mathit{icmp\_echo}, \mathit{dst\_host} = \mathit{host}_B)$	02100
$(\mathit{flag} = \mathit{S0}, \mathit{service} = \mathit{http}, \mathit{src\_host} = \mathit{host}_A)$	21010
$(\mathit{service} = \mathit{user\_app}, \mathit{src\_host} = \mathit{host}_A)$	03010
$(\mathit{flag} = \mathit{SF}, \mathit{service} = \mathit{icmp\_echo}, \mathit{dst\_host} = \mathit{host}_B, \mathit{src\_host} = \mathit{host}_C)$	12120
...	...

Table 6.2: Encodings of Associations for Comparison

network connection records for comparison purposes, we use the following decreasing "order of importance": *flag*, axis attribute, reference attribute, the rest of "essential" attributes in alphabetical order, and the remaining attributes in alphabetical order. The attribute *flag* is most important (i.e., interesting) in an association since its value is a summary of how the connection has behaved according to the network protocols. Any value other than "SF" (i.e., normal connection establishment and termination) is of great interest for intrusion detection.

For pattern comparison, we first convert the 3-d encoding of an episode into a 1-d number. Assume $\mathit{encoding}_X = x_1x_2...x_n$, $\mathit{encoding}_Y = y_1y_2...y_n$, and $\mathit{encoding}_Z = z_1z_2...z_n$, then the 1-d representation is

$$x_1z_1y_1x_2z_2y_2 \ldots x_nz_ny_n$$

This presentation preserves the "order of importance" of attributes (in association encoding) and considers the rule structure of an episode. Here two episodes that have similar first "body" (i.e., X) and "head" (i.e., Z) will be mapped to closer numbers. As an example, using the association encodings in Table 6.2 and "ignoring" *dst_host*, *src_host*, and *src_bytes*, the "syn flood" pattern

$$\begin{aligned}&(\mathit{service} = \mathit{http}, \mathit{flag} = \mathit{S0}),\\ &(\mathit{service} = \mathit{http}, \mathit{flag} = \mathit{S0}) \rightarrow\\ &(\mathit{service} = \mathit{http}, \mathit{flag} = \mathit{S0})\end{aligned}$$

is encoded as 222111. Similarly, a "normal" pattern,

$$\begin{aligned}&(\mathit{flag} = \mathit{SF}, \mathit{service} = \mathit{http}),\\ &(\mathit{flag} = \mathit{SF}, \mathit{service} = \mathit{icmp\_echo}) \rightarrow\\ &(\mathit{flag} = \mathit{SF}, \mathit{service} = \mathit{http})\end{aligned}$$

is encoded as 111112.

When comparing two episodes using their 1-d numbers, a simple digit-wise comparison is performed. That is, in the resulting *diff* score

$$d_{x_1} d_{z_1} d_{y_1} d_{x_2} d_{z_2} d_{y_2} \ldots d_{xn} d_{zn} d_{yn}$$

each digit, e.g., d_{x_i}, is the absolute value difference in the corresponding digit, e.g., x_i, of the two episodes. For example, when comparing the "syn flood" pattern with the "normal" pattern, the *diff* score is 111001.

Given the normal patterns and patterns from an intrusion dataset, we can identify the intrusion-only patterns using the procedure outlined in Figure 6.11.

Here step (1) encodes pattern sets from two datasets, e.g., one from the the normal audit dataset and the other from the intrusion audit dataset. In step (2), for each pattern from dataset 2, e.g., the intrusion dataset, the procedure calculates its *diff* score with each pattern of dataset 1, e.g., the normal dataset, and keeps the lowest *diff* score as the "unique" score, e.g., the "intrusion" score, for this pattern. Step (3) outputs all patterns that have a user-specified top percentage of patterns

```
Input: a set of patterns $P_1$ from dataset 1,
       a set of patterns $P_2$ from dataset 2,
       and a user-specified $p$ percent value;
Output: a set of top $p$ percent unique patterns $P_u$
        that are in $P_2$ but not in $P_1$;
Begin
(1)   Encode all patterns:
          $E_1$ is the encodings of $P_1$, and $E_2$ is the encodings of $P_2$;

(2)   for each $e_2 \in E_2$ do begin
          $unique_2 = \infty$;
          for each $e_1 \in E_1$ do begin
              $diff_2 = diff$ score between $e_2$ and $e_1$;
              if $diff_2 < unique_2$ then
                  $unique_2 = diff_2$;
          end
      end

(3)   sort all $e_2 \in E_2$ according to its $unique_2$;
      $P_u$ = patterns in $P_2$ that correspond to the
          top $p$ percent $e_2$ encodings that have non-zero $unique_2$ scores;
end
```

Figure 6.11: Identifying Unique Patterns

with the highest non-zero unique scores. As an example, since there is no normal pattern with *flag* = *S0* in all its itemsets, the *diff* score for the "syn flood" pattern, e.g., 111001, is very high, and thus the pattern will be selected as one of the top intrusion-only patterns.

When used to identify intrusion-only patterns, this procedure considers a pattern from the intrusion dataset as "normal" as long as it has a match with one of the normal patterns. For simplicity, we omit the comparisons on the support and confidence values once the heads and bodies of the rules match. We have not encountered a case where two matched rules have values that are more than 5% apart from each other, which is considered an acceptable threshold.

6.3.2 Feature Construction

Each of the intrusion-only patterns can be used for constructing additional features into the connection records, using the algorithm in Figure 6.12. This procedure parses a frequent episode and uses three simple operators, *count*, *percent*, and *average*, to construct simple statistical features. These features are also temporal since they measure only the connections that are within a time window w and share the same reference attribute value. The intuition behind this feature construction algorithm comes from the straightforward interpretation of a frequent episode. For example, if the same attribute value appears in all the itemsets of an episode, then there is a large percentage of records (i.e., the original data) that have the same value.

We treat the "essential" and "non-essential" attributes differently. The "essential" attributes describe the anatomy of an intrusion which normally involves a sequence of connections, for example, "the same *service* (i.e., *port*) is targeted". The actual values, e.g., "http", is often not important because the same attack method can be applied to different targets, e.g., "ftp". On the other hand, the actual "non-essential" attribute values, e.g., *flag* = *S0*, often indicate the invariant of an intrusion regardless of the actual targets, because they summarize the behavior according to the network protocols.

As an example, the "syn flood" pattern results in the following additional features: a count of connections to the same *dst_host* in the past 2 seconds, and

```
Input: a frequent episode, and the set of existing features
       in connection records, $\mathcal{F}$
Output: the updated $\mathcal{F}$
Begin
(1)     Let $F_0$ (e.g., dst_host) be the reference attribute used to
            mine the episode;
(2)     Let $w$, in seconds, be the minimum width of the episode;
        /* all the following features consider only the connections in past $w$
         * seconds that share the same value in $F_0$ as the current connection
         */
(3)     Let $count\_same_{F_0}$ be the number of these connections;
(4)     $\mathcal{F} = \mathcal{F} \cup \{count\_same_{F_0}\}$;
(5)     for each "essential attribute" $F_1$ other than $F_0$ do begin
(6)         if the same $F_1$ value is in all the item sets then begin
(7)             Let $percent\_same_{F_1}$ be the percentage of connections that share
                    the same $F_1$ value as the current connection;
(8)             $\mathcal{F} = \mathcal{F} \cup \{percent\_same_{F_1}\}$;
            end else
                /* there are different $F_1$ or no $F_1$ values at all */
(9)             Let $percent\_diff_{F_1}$ be the percentage of different $F_1$ values
                    in the connections;
(10)            $\mathcal{F} = \mathcal{F} \cup \{percent\_diff_{F_1}\}$;
            end
        end
(11)    for each value $V_2$ of an "non-essential" attribute $F_2$ do begin
(12)        if $V_2$ is in all the item sets then begin
(13)            Let $percent\_same_{V_2}$ be the percentage of connections that share
                    the same $V_2$ value as the current connection;
(14)            $\mathcal{F} = \mathcal{F} \cup \{percent\_same_{V_2}\}$;
            end
(15)        else if $F_2$ is a numerical attribute then begin
(16)            Let $average_{F_2}$ be the average of the $F_2$ values of the connections;
(17)            $\mathcal{F} = \mathcal{F} \cup \{average_{F_2}\}$;
            end
        end
end
```

Figure 6.12: Constructing Features from Frequent Episode

among these connections, the percentage of those that have the same *service* as the current, and the percentage of those that have the "S0" *flag*.

For efficiency, the algorithm can operate on the encoding of the input episode. Parsing and checking attribute values then become trivial digit comparisons. The procedure in Figure 6.11 need to also output the encodings of the intrusion-only patterns.

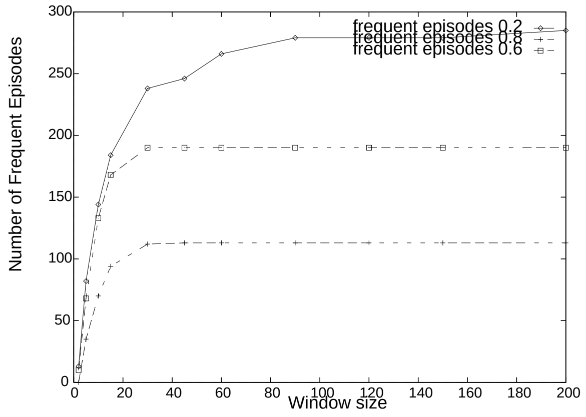


Figure 6.13: Effects of Window Sizes on the Number of Frequent Episodes.

An open problem here is how to decide the right time window value w. Our experience showed that when we plot the number of patterns generated using different w values, the plot tends to stabilize after the initial sharp jump. We call the smallest w in the stable region w_0. Our experiments showed that the plot of accuracies of the classifiers that use the temporal and statistical features calculated with different w, also stabilizes after $w \geq w_0$ and tend to taper off. Figure 6.13 plots the number of episodes mined from the normal *in-coming* traffic data that is part of the experiments described in Section 4.3. We mined the frequent episodes using different support thresholds. We can see that the number of frequent patterns increases sharply as w goes from 2s to 30s, it then gradually stabilizes[2]. This phenomenon coincides with the effects of time window sizes on classifier accuracy in Figure 4.2.

Intuitively, a requirement for the window size is that its set of sequential patterns is stable, that is, sufficient patterns are captured and noise is small. We therefore use w_0 for adding temporal and statistical features.

6.3.3 Theoretical Underpinnings

In Chapter 4 we stated that the main focus of our research is in feature selection and construction. The experiments described in Section 4.3 showed that temporal and statistical features need to be constructed and included in the classification models. In Chapter 5 we proposed to use the frequent patterns mined from audit records as guidelines for feature construction, and described the basic algorithms for mining audit data and their extensions. In this chapter, we outlined the algorithms for comparing and identifying patterns, and for constructing features from the intrusion-only patterns. We now explain why the additional features constructed from the frequent patterns can improve the accuracy of the classifiers.

Recall in Chapter 4, we pointed out that machine learning algorithms select features (attributes) into a classification model based on their information gain measures. Here we show that the features constructed from the frequent patterns are likely to have higher information gains than existing features.

[2]By the nature of the frequent episodes algorithm, the number of episodes can only increase as w increases.

Intrusion-only Patterns Are the Results of Intrusion Records

Because of the nature of many network attacks, each connection needs to be examined not only with its sequence of packets, but also within the context of a sequence of related connections. We have argued that the frequent sequential patterns, computed using our extended algorithms, capture the temporal and statistical characteristics of network connections. We now show that the intrusion-only patterns are the results of the intrusion connection records in the audit dataset.

First, we have the following straightforward theorem:

Theorem 6.1 *Given two sets of patterns from dataset 1 and dataset 2, if there are unique patterns from dataset 2 according to Algorithm 6.11, then "statistically speaking", the dataset contains unique records or unique sequences of records.*

Proof:

First note that the two sets of patterns are computed from two datasets using the same data mining strategies, i.e., the same algorithm with the same support, confidence, and window requirements, and axis and reference attributes.

According to Algorithm 6.7, the definition of *diff* score, and Algorithm 6.11, a pattern from dataset 2, e.g.,

$$L_1, L_2 \rightarrow R$$

can be unique, i.e., with non-zero *unique* score when compared with patterns from dataset 1, if

- at least one of its itemsets (i.e., association), e.g., L_2, is unique. In such case, there are records in dataset 2 that contain the items of this association; however, "statistically", i.e., with regard to the same support and confidence thresholds, there are no such records in dataset 1. Otherwise, the same association would have been computed from dataset 1.

- the frequent episode is unique; in such case, there are sequences of records that contain the itemsets (i.e., associations) of the episode from dataset 2; however, "statistically and temporally", i.e., with regard to the same support, confidence, and window requirements, there are no such sequences of records in dataset 1. Otherwise, the same frequent episode would have been computed from dataset 1.

We need to emphasize here that the uniqueness of records or sequences of records may not be valid without the statistical references, i.e., the support, confidence, and window requirements. For example, assume that dataset 2 has a total m records, and among them, n records contain items of L_2. Further,

$$\frac{n}{m} \geq min\_support$$

Dataset 1 can also contain n records that contain items of L_2, but if it has a total of $k * m$ records, and

$$\frac{n}{k * m} < min\_support$$

L_2 then becomes a unique association of dataset 2. However, we can not claim that the n records in dataset 2 are unique. ▮

Given the aggregate normal pattern set and a set of patterns from audit data of a (simulated) network intrusion, Algorithm 6.11 outputs the intrusion-only patterns. According to Theorem 6.1, the intrusion-only patterns indicate that the intrusion audit data has "statistically and temporally" unique connections or sequences of connections compared with normal network traffic. These connections are mostly the intrusion connections. That is, although some "unique" patterns from the intrusion dataset may actually be the result of unique normal connections, the designs of our data mining, and pattern encoding and comparison algorithms ensure that most of the intrusion-only patterns corresponds to actual intrusion connections, and vice versa, most intrusion connections are captured in the intrusion-only patterns.

The Constructed Features Have Higher Information Gains

The feature construction algorithm described in Figure 6.12 can be used to parse each intrusion-only pattern and add a number of temporal and statistical features to each connection record. These features describe the anatomy and invariant information of the intrusion pattern. They are temporal because they are measures on connections within a time window. They are statistical because they are simple statistical calculations, i.e., *count*, *percent*, and *average* on existing features.

For the features constructed from an intrusion-only pattern, the feature values in the intrusion connection records that are responsible for the pattern will be very different from the feature values in the normal connection records. This is precisely the purpose of adding these features. For example, for the feature from the "syn flood" pattern, *for the connections to the same destination host, the percentage that have S0 flag*, normal records have values close to 0, but "syn flood" records have values above 80%. These features have high information gain because their value ranges can separate intrusion records from the normal records.

We need to appreciate the fact that these constructed features normally have higher information gains than the existing features, i.e., they can be more useful in building a classification model. For example, *flag* = *S0* can appear in both normal connections and intrusion connections. Therefore, the feature *flag* does not have a high information gain. From Theorem 6.1, a intrusion-only pattern suggests the existence of unique sequences of connection records, only in the "temporal and statistical" sense, i.e., when support, confidence, and time window are considered. Therefore, the temporal and statistical features, e.g., those that are based on *flag*, capture the "uniqueness" of the intrusion connections more accurately, and thus have higher information gains.

6.4 Summary

In this chapter, we discussed how to utilize the mined frequent patterns for audit data gathering and feature construction. We based our algorithms on the belief that the mined patterns are the statistical summaries of the system behavior.

We first described how to incrementally merge patterns mined from multiple audit datasets into an aggregate rule set. We suggested that when the aggregate rule set begins to stabilize, the aggregate audit dataset is "sufficient" for learning classification models. The results from the experiments of building anomaly detection models on the LBL dataset confirmed our hypothesis.

We presented a pattern encoding scheme that is based on user-defined "order of importance" of audit data attributes. This encoding scheme facilitates pattern visualization and analysis. The experiments on the IWSS16 dataset showed that when patterns are encoded and visualized in 3-D plots, the intrusion patterns can be easily identified because they formed distinct clusters.

We then outlined a pattern comparison algorithm that automates the process of identifying intrusion-only patterns. The *diff* measure between two patterns is calculated based on the pair-wise difference of their encodings. The patterns from an intrusion dataset that have high *diff* scores, when compared with the aggregate normal patterns, are considered intrusion-only patterns. We discussed an feature construction algorithm that parses each intrusion-only pattern and adds temporal and statistical features according to the anatomy and invariant information of the intrusion pattern.

We explained the theoretical underpinnings of our feature construction process. We showed that the intrusion-only patterns are the results of intrusion connections, and the features from these patterns have higher information gain measures because their values separate intrusion records from normal records. Therefore, these features are more useful in building accurate classification models.

In Chapter 7, we described our extensive experiments on the audit data from the 1998 DARPA Intrusion Detection Evaluation program. These experiments demonstrated and verified the utilities of the algorithms and tools of MADAM ID.

Chapter 7

Putting It All Together: Experiments in the 1998 DARPA Intrusion Detection Evaluation

In this chapter, we describe our experiments in building intrusion detection models on the audit data from the 1998 DARPA Intrusion Detection Evaluation Program. In these experiments, we applied the algorithms and tools of MADAM ID to process audit data, mine patterns, construct features, and build RIPPER classifiers.

We first describe the experiments on *tcpdump* data. The results of these experiments were submitted to DARPA and were evaluated by MIT Lincoln Labs. We then report recent experiments on *BSM* data, which were performed after the DARPA evaluation. We discuss our experiences and evaluate the strengths and weaknesses of MADAM ID.

7.1 Experiments on *tcpdump* Data

We participated in the DARPA Intrusion Detection Evaluation Program, prepared and managed by MIT Lincoln Labs. The objective of this study was to survey and evaluate the state of the art in intrusion detection research. A standard set of extensively gathered audit data, which includes a wide variety of intrusions simulated in a military network environment, is provided by DARPA. Each participating site was required to build intrusion detection models or tweak their existing system parameters using the training data, and send the results i.e., detected intrusions on the test data back to DARPA for performance evaluation. We report our experience here.

7.1.1 The DARPA Data

We were provided with about 4 gigabytes of compressed *tcpdump* data of 7 weeks of network traffic. This data can be processed into about 5 million of connection records of about 100 bytes each. The data contains content (i.e., the data portion) of every packet transmitted between hosts inside and outside a simulated military base. BSM audit data from one UNIX Solaris host for some network sessions were also provided.

Four main categories of attacks were simulated, they are:

- DOS, denial-of-service, for example, ping-of-death, teardrop, smurf, syn flood, etc.,
- R2L, unauthorized access from a remote machine, for example, guessing password,

- U2R, unauthorized access to local superuser privileges by a local unprivileged user, for example, various of buffer overflow attacks,
- PROBING, surveillance and probing, for example, port-scan, ping-sweep, etc.

In addition, there were anomalous user behavior such as "a manager becomes (i.e., behaves like) a system administrator".

Data Pre-processing

We used Bro [Paxson, 1998] as the packet filtering and reassembling engine. We extended Bro to handle ICMP packets, and made changes to its packet fragment inspection modules since it crashed when processing data that contains teardrop or ping-of-death attacks.

We implemented a Bro "connection finished" event handler to output a summarized record for each connection. Each connection record includes a set of "intrinsic" features shown in Table 7.1.

7.1.2 Misuse Detection

The training data from DARPA includes "list files" that identify the timestamp, source host and port, destination host and port, and the name of each attack. We used this information to select intrusion data to perform pattern mining and feature construction, and to label each connection record with "normal" or an attack type to create training data for building classification models.

Since the amount of audit data is huge, for example, some days have several millions of connection records due to some nasty DOS attacks, we did not aggregate

feature	**description**	**value type**
duration	length (number of seconds) of the connection	continuous
protocol_type	type of the protocol, e.g. tcp, udp, etc.	discrete
service	network service on the destination, e.g., http, telnet, etc.	discrete
src_bytes	number of data bytes from source to destination	continuous
dst_bytes	number of data bytes from destination to source	continuous
flag	normal or error status of the connection	discrete
land	1 - connection is from/to the same host/port; 0 - otherwise	discrete
wrong_fragment	number of "wrong" fragments	continuous
urgent	number of urgent packets	continuous

Table 7.1: Intrinsic Features of Network Connection Records

all the connection records into a single training data set. Instead, we extracted all the connection records that fall within a surrounding time window of plus and minus 5 minutes of the whole duration of each attack to create a data set for each attack type. We also randomly extracted sequences of normal connections records to create the normal data set.

Manual and Automatic Feature Construction

Following the feature construction approach described in Section 6.3, for each attack type, e.g., syn flood, port-scan, etc., we performed pattern mining and comparison using its intrusion data set and the normal data set. Since for each attack method, the actual network hosts are irrelevant and there were over a thousand different hosts in the DARPA data, we "post-processed" the frequent patterns before encoding and comparison, using the following procedure:

- Examine each itemset (from LHS to RHS) of a frequent pattern one by one
  - replace the first *src_host* value encountered by s_0; likewise, replace the first *dst_host* value by d_0;
  - when a *src_host* value is encountered, check if it is the same as one of the previous source hosts in the pattern; if yes, replace it with the appropriate s_i; otherwise replace it with s_{n+1}; perform the same processing on a *dst_host* value.

For example, a pattern

$$(service = http, src\_host = host_A),$$
$$(service = telnet, dst\_host = host_B) \rightarrow$$
$$(service = smtp, src\_host = host_C, dst\_host = host_B), [0.2, 0.1, 2s]$$

is post-processed into

$$(service = http, src\_host = s_0),$$
$$(service = telnet, dst\_host = d_0) \rightarrow$$
$$(service = smtp, src\_host = s_1, dst\_host = d_0), [0.2, 0.1, 2s]$$

The number of unique patterns within a pattern set is significantly reduced as a result of the post-processing. The processes of creating an aggregate normal pattern set, and pattern encoding and comparison all became much more efficient.

We constructed appropriate features according to the top 20% intrusion-only patterns of each attack type. Here we summarize the temporal and statistical features automatically constructed by our system:

- The "*same host*" feature that examine only the connections in the past 2 seconds that have the same destination host as the current connection:

  - the count of such connections, the percentage of connections that have the same service as the current one, the percentage of different services, the percentage of SYN errors, and the percentage of REJ (i.e., rejected connection) errors;

- The "same service" features that examine only the connections in the past 2 seconds that have the same service as the current connection:

  - the count of such connections, the percentage of different destination hosts, the percentage of SYN errors, and the percentage of REJ errors.

We call these the (time-based) "traffic" features for connection records. They are summarized in Table 7.2.

There are several "slow" PROBING attacks that scan the hosts (or ports) using a much larger time interval than 2 seconds, for example, one in every minute. As a result, these attacks did not produce intrusion-only patterns with the time window of 2 seconds. We sorted these connection records by the destination hosts, and applied the same pattern mining and feature construction process. Instead of using a time window of 2 seconds, we now used a "connection" window of 100 connections, and constructed a mirror set of "host-based traffic" features as the (time-based) "traffic" features.

We discovered that unlike most of the DOS and PROBING attacks, the R2L and U2R attacks don't have any intrusion-only frequent patterns. This is because most of the DOS and PROBING attacks involve sending a lot of connections to some host(s) in a very short period of time, and therefore can have frequent sequential patterns that are different from the normal traffic. The R2L and U2R attacks are

feature	**description**	**value type**
count	number of connections to the same host as the current connection in the past 2 seconds	continuous
	the following features refer to these same-host connections	
serror_%	% of connections that have "SYN" errors	continuous
rerror_%	% of connections that have "REJ" errors	continuous
same_srv_%	% of connections to the same service	continuous
diff_srv_%	% of connections to different services	continuous
srv_count	number of connections to the same service as the current connection in the past 2 seconds	continuous
	the following features refer to these same-service connections	
srv_serror_%	% of connections that have "SYN" errors	continuous
srv_rerror_%	% of connections that have "REJ" errors	continuous
srv_diff_host_%	% of connections to different hosts	continuous

Table 7.2: Traffic Features of Network Connection Records

embedded in the data portions of the packets and normally involve only a single connection. Therefore, it is unlikely that they can have any unique frequent traffic patterns. In other words, our automatic feature construction process would fail to produce any features for these attacks.

After studying the outcome of this mining process, we focussed our attention to the content of the connections. Our current data mining algorithms cannot deal with unstructured data contents of the IP packets. We instead relied on domain knowledge to define suitable features. In the Bro event handlers, we added functions that inspect data exchanges of interactive TCP connections (e.g., telnet,

feature	**description**	**value type**
hot	number of "hot indicators"	continuous
failed_logins	number of failed login attempts	continuous
logged_in	1 - successfully logged in; 0 - otherwise	discrete
compromised	number of "compromised" conditions	continuous
root_shell	1 - root shell is obtained; 0 - otherwise	discrete
su	1 - "su root" command attempted; 0 - otherwise	discrete
file_creations	number file creation operations	continuous
shells	number of shell prompts	continuous
access_files	number of write, delete, and create operations on access control files	continuous
outbound_cmds	number of outbound commands in a ftp session	continuous
hot_login	1 - the login belongs to the "hot" list (e.g., *root*, *adm*, etc.); 0 - otherwise	discrete
guest_login	1 - the login is a "guest" login (e.g., *guest*, *anonymous*, etc.) ; 0 - otherwise	discrete

Table 7.3: Content Features of Network Connection Records

ftp, smtp, etc.). These functions assign values to a set of "content" features to indicate whether the data contents suggest suspicious behavior. These features are: number of failed logins, successfully logged in or not, whether logged in as root, whether a root shell is obtained, whether a su command is attempted and succeeded, number of access to access control files (e.g., "/etc/passwd", ".rhosts"), number of compromised states on the destination host (e.g., file/path "not found" errors, and "Jump to" instructions, etc.), number of hot indicators, (e.g., access to system directories, creation and execution of programs, etc.), and number of outbound connections during a *ftp* session. These features are summarized in Table 7.3. Our approach here is to include an extensive set of indicators, and then let classification programs decide which minimal set of discriminating features should be used to identify intrusions.

label	service	flag	count	srv_count	rerror_%	diff_srv_%	...
normal	ecr_i	SF	1	1	0	1	...
smurf	ecr_i	SF	350	350	0	0	...
satan	user-level	REJ	231	1	85%	89%	...
normal	http	SF	1	0	0	1	...
...	...	...	...	...	...	...	...

Table 7.4: Example "Traffic" Connection Records

RIPPER rule	Meaning
smurf :- count $\geq$ 5, srv_count $\geq$ 5, service = ecr_i.	If the service is icmp echo request, and for the past 2 seconds, the number of connections that have the same destination host as the current one is at least 5, and the number of connections that have the same service as the current one is at least 5, then this is a smurf attack (a DOS attack).
satan :- rerror_% $\geq$ 83%, diff_srv_% $\geq$ 87%.	If for the connections in the past 2 seconds that have the same destination host as the current connection, the percentage of rejected connections is at least 83%, and the percentage of different services is at least 87%, then this is a satan attack (a PROBING attack).

Table 7.5: Example RIPPER Rules for DOS and PROBING Attacks

Detection Models

It is evident that different categories of intrusions require different sets of constructed features for detection purposes. We therefore built classification models using different feature sets:

- The "traffic" model: each connection record contains the "intrinsic" and the "traffic" features. Table 7.4 shows some example labeled connection records. The resultant RIPPER classifier detects the DOS and PROBING attacks. Table 7.5 shows some example RIPPER rules.

- The "host-based traffic" model: each connection record contains the "intrinsic" and the host-based "traffic" features. The resultant RIPPER classifiers detect the slow PROBING attacks.

- The "content" model: each connection record contains the "intrinsic" and the "content" features. Table 7.6 shows some example labeled connection records. The resultant RIPPER classifier detects the R2L and U2R attacks. Table 7.7 shows some example RIPPER rules.

label	service	flag	hot	failed_logins	compromised	root_shell	su	...
normal	ftp	SF	0	0	0	0	0	...
normal	telnet	SF	0	0	0	3	1	...
guess	telnet	SF	0	6	0	0	0	...
normal	telnet	SF	0	0	0	0	0	...
overflow	telnet	SF	3	0	2	1	0	...
normal	rlogin	SF	0	0	0	0	0	...
guess	telnet	SF	0	5	0	0	0	...
overflow	telnet	SF	3	0	2	1	0	...
normal	telnet	SF	0	0	0	0	0	...
...	...	...	...	...	...	...	...	...

Table 7.6: Example TCP Connection Records

RIPPER rule	Meaning
guess :- failed_logins >= 5.	If number of failed logins is greater than 5, then this telnet connection is "guess", a guessing password attack.
overflow :- hot = 3, compromised = 2, root_shell = 1.	If the number of hot indicators is 3, the number of compromised conditions is 2, and a root shell is obtained, then this telnet connection is a buffer overflow attack.
...	...
normal :- true.	If none of the above, then this connection is "normal".

Table 7.7: Example RIPPER Rules for R2L and U2R Attacks

These classification models each specialize to a certain type of intrusion. We then constructed a meta-level classifier to combine these detection models. Each meta-level training record consists of four features, the three predictions each from one of the base models, plus the true class label (i.e., "normal" and an attack type). RIPPER was then applied to learn the rules that combine the evidence from the "traffic", host-based "traffic", and "content" classifiers to make a (final) prediction on a connection. The resulting meta-level rules basically use the predictions from the "content" model to detect R2L and U2R attacks, and the combination of "traffic" and host-based "traffic" models to detect the DOS and (fast and slow) PROBING attacks. That is, the meta-classifier predicts a connection as an attack of R2L or U2R whenever the "content" model does so; and an attack of DOS or PROBING whenever the "traffic" model does so, or whenever the "traffic" model predicts "normal" but the host-based model predicts a PROBING attack.

Model	# of features in records	# of rules	# of features in rules
content	22	55	11
traffic	20	26	4+**9**
host traffic	14	8	1+**5**

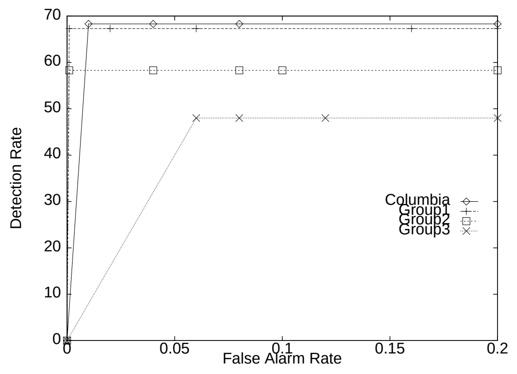
Table 7.8: Model Complexities

Table 7.8 summarizes the complexity of the base models in terms of the number of features in a connection record, the number of RIPPER rules produced, and the number of distinct features actually used in the rules. The numbers in bold, for example, **9**, indicate the number of automatically constructed temporal and statistical features being used in the RIPPER rules. We see that for both the "traffic" and "host-based traffic" models, our feature construction process contribute the majority of the features. We should point out that not all features in the connection records were selected by RIPPER. This is because RIPPER, like most classification algorithms, has a built-in "feature selection" process to select the most discriminating and generalizable features according to their statistical significance and performance (e.g., in a hold-out test dataset that simulates the "unseen/future" data). Because of the large amount of audit data, a human expert is not able to manually gather and test various statistics, and thus tend to do a poor job in selecting the features. As a result, hand crafted "signature" rules tend to be very specific to a small intrusion data set. Alternative classification algorithms that compute underlying probability distributions may indeed require all features be evaluated in their resultant models. A crucial issue here is the tradeoff between model accuracy and model cost. The RIPPER output indicates that some features are irrelevant and hence we need not compute these at run-time, thus reducing the cost of detection. This is the subject matter of our ongoing research.

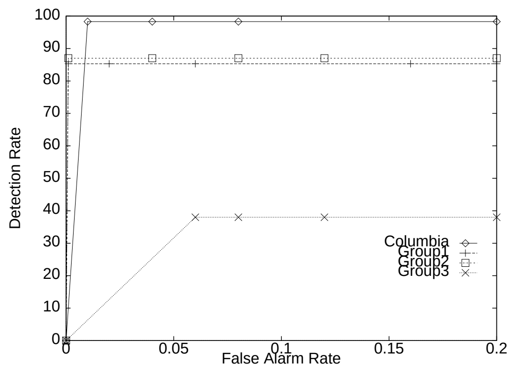
Results

We report the performance of our detection models as evaluated by MIT Lincoln Labs. We trained our intrusion detection models, i.e., the base models and the meta-level classifier, using the 7 weeks of labeled data, and used them to make predictions on the 2 weeks of unlabeled test data (i.e., we were not told which connection is an attack). The test data contains a total of 38 attack types, with 14 types in test data only (i.e., our models were not trained with instances of these attack types).

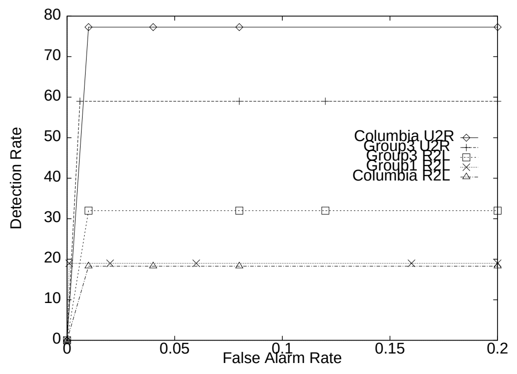
Figure 7.1 shows the ROC curves of the detection models by attack categories as well as on all intrusions. In each of these ROC plots, the x-axis is the



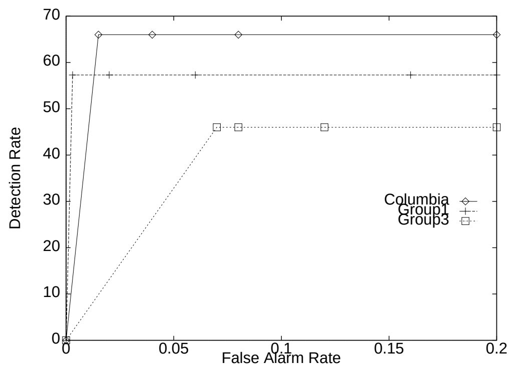
(a) DOS



(b) PROBING



(c) U2R and R2L



(d) Overall

Figure 7.1: Performance of *tcpdump* Misuse Detection Models: ROC Curves on Detection Rates and False Alarm Rates

false alarm rate, calculated as the percentage of normal connections classified as an intrusion; the y-axis is the detection rate, calculated as the percentage of intrusions detected. A data point in the upper left corner corresponds to optimal performance, i.e., high detection rate with low false alarm rate. We compare here our models with other participants (denoted as Group 1 through 3) in the DARPA evaluation program[1]. These participating groups used knowledge engineering ap-

[1]The tested systems produced binary output, hence, the ROC's are not continuous. These plots are duplicated from the presentation slides of a report given by Lincoln Labs in a DARPA PI meeting. The slides can be viewed on line via `http://www.cs.columbia.edu/~sal/JAM/PROJECT/MIT/mit-index.html`.

proaches to build their intrusion detection models. We can see from the figure that our detection model has the best overall performance, and in all but one attack category, our model is one of the best two. However, it is also clear that all models performed very poorly on R2L attacks. For all intrusions, an overall detection rate of below 70% is hardly satisfactory in a mission critical environment.

Category	Old	New
DOS	79.9	24.3
PROBING	97.0	96.7
U2R	75.0	81.8
R2L	60.0	5.9
Overall	80.2	37.7

Table 7.9: Comparing Detection Rates (in %) on Old and New Attacks

Although our models were intended for misuse detection, we had hoped that the features we constructed would be general enough so that the models can detect new variations of the known intrusions. Table 7.9 compares the detection rates of old intrusions and new intrusions. Here new intrusions refer to those that did not have corresponding instances in the training data. We see that our models were able to detect a large percentage of new PROBING and U2R attacks, but not as effective for new DOS and R2L attacks.

Discussion

PROBING attacks have relatively limited variance because they all involve making connections to a large number of hosts or ports in a given time frame. Likewise, the outcome of all U2R attacks is that a root shell is obtained without legitimate means, e.g., login as root, su to root, etc. Thus, for these two categories of attacks, given some representative instances in the training data, our data mining system was able to construct features that capture their general behavior patterns. As a result, our detection models can detect a high percentage of old and new PROBING and U2R attacks. On the other hand, DOS and R2L have a wide variety of behavior because they exploit the weaknesses of a large number of different network or system services. The features constructed based on the available attack instances are very specialized to the known attack types. Our detection models therefore missed a large number of new DOS and R2L attacks.

The results here are not entirely surprising since our models are *misuse* detection models. We need to use anomaly detection models on network traffic or system programs to guard against the new and diversified attacks. Anomaly detection is much more challenging than misuse detection. For example, we need to first decide whether we should build normal profile for each network service or a group of services, and for each host or a groups of hosts. The feature construction process will likely to be more complex since unlike a relatively small number of intrusion-only patterns, normal network traffic can have a large number of variations. Network anomaly detection is an important problem and an active area of research.

7.1.3 User Anomaly Detection

"Insiders" misusing their privileges can be hard to detect since they don't normally need to break-in, and IDSs and security personnel tend to focus on guarding outside attacks. Insider problems are some of the most vexing problems for security personnel. (Indeed, who checks the checkers, i.e., the person to whom the IDS reports?)

It is often very difficult to classify a single event by a user as normal or abnormal because the unpredictable nature of most people. A user's actions during a login session needs to be studied as a whole to determine whether he or she is behaving normally.

time	hostname	command	arg1	arg2
am	pascal	mkdir	dir1	
am	pascal	cd	dir1	
am	pascal	vi	tex	
am	pascal	tex	vi	
am	pascal	mail	fredd	
am	pascal	subject	progress	
am	pascal	vi	tex	
am	pascal	mail	williamf	
am	pascal	subject	progress	
	...	...	...	...

Table 7.10: Shell Command Records

We used Bro event handlers to examine the telnet sessions, and extract the shell commands of the users. We further pre-processed the shell commands by replacing timestamps with am, pm, and nt (for night), and eliminated the input (i.e., contents) of edit and sendmail commands, and kept only the filename extensions. Table 7.10 shows examples of the processed command data. These shell command records were used for user anomaly detection.

Our initial exploratory approach is to mine the frequent patterns from the command data, and merge or add the patterns into an aggregate set to form the normal usage profile of a user. A new pattern can be merged with an old pattern if they have the same left-hand-sides and right-hand-sides, their support values are within a 5% of each other, and their confidence values are also within 5% of each other.

To analyze a user login session, we mine the frequent patterns from the sequence of commands during this session. This new pattern set is compared with the profile pattern set and a *similarity* score is assigned. Assume that the new set has n patterns and among them, there are m patterns that have "matches" (i.e., rules that they can be merged with) in the profile pattern set, then the similarity score is simply $\frac{m}{n}$. Obviously, a higher similarity score means a higher likelihood that the user's behavior agrees with his or her historical profile.

User	Normal Activities
sysadm	logs in as root, cats the password file, and runs commands such as top.
programmer1	writes public domain C code, uses a vi editor, compiles the C code, reads and sends mails, and executes unix commands.
programmer2	a similar user profile, but works in afternoons and evenings.
secretary	edits latex files, runs latex, reads mails, and sends mails.
manager1	reads and sends mails
manager2	reads mails.

Table 7.11: User Descriptions

The DARPA data also includes user anomaly data to evaluate anomaly detection systems. Table 7.11 describes the consistent behavior of the 6 users for anomaly analysis. Note that since we were the only group that performed anomaly detection on the test data, Lincoln Labs did not evaluate our results. We report our experiments on the training data here.

We applied our frequent episode algorithms to the command data from each login session (of the same user), with *command* as the axis feature and $w = 5$ (i.e., we look for patterns within the range of 5 consecutive commands), to mine

User	Anomaly Description
programmer2	logs in from beta
secretary	logs in at night
sysadm	logs in from jupiter
programmer1	becomes a secretary
secretary	becomes a manager
programmer1	logs in at night
sysadm	**becomes a programmer**
manager1	becomes a sysadm
manager2	logs in from pluto

Table 7.12: User Anomaly Description

User	Normal	Anomaly
programmer2	(0.58, 0.79)	0.00
secretary	(∞, ∞)	0.00
sysadm	(0.84, 0.95)	0.00
programmer1	(0.31, 1.00)	0.04
secretary	(0.41, 0.98)	0.17
programmer1	(∞, ∞)	0.00
sysadm	**(0.64, 0.95)**	**0.00**
manager1	(0.57, 1.00)	0.00
manager2	(1.00, 1.00)	0.00

Table 7.13: Similarity with User's Own Profile

the frequent sequential patterns on the associations among user commands, their arguments, time segments, and hosts. We used the first 4 weeks as a data gathering period, during which we simply merged the patterns into each user's profiles. Each user has 3 profiles, one for the activities of each time segment (am, pm, and nt). We used the 5th week as the training period, during which we compared the patterns from each session to the profile of the time segment. We record the normal range of the similarity scores during this week. The data in the 6th week has some user anomalies, as described in Table 7.12. For each of the anomalous sessions, we compared its patterns against the original user's profile, and then compared the resulting similarity score against the recorded normal range of the same time segment. In Table 7.13, the column labeled "Normal" is the range of similarity of each user against his or her own profile as recorded during the 5th week. A ∞ here means that the user did not login during the time segment in the 5th week. The column "Anomaly" is the similarity measure of the anomalous session described in Table 7.12. We see that all anomalous sessions can be clearly detected since their similarity scores are much smaller than the normal range. For example, when the sysadm becomes programmer, his/her patterns have zero matches with the sysadm's profile; while for the whole 5th week, the pm similarity scores are in the range of 0.64 to 0.95.

	Normal				**Anomaly**			
User	**P**	**S**	**M**	**SA**	**P**	**S**	**M**	**SA**
p2	(0.33, 0.71)	(0.04, 0.11)	(0.00, 0.03)	(0.00, 0.07)	0.00	0.00	0.00	0.00
s	(∞, ∞)	(∞, ∞)	(∞, ∞)	(∞, ∞)	0.04	0.00	0.00	0.00
sa	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)	(0.51, 0.81)	0.00	0.00	0.00	0.00
p1	(0.12, 0.57)	(0.06, 0.09)	(0.00, 0.00)	(0.04, 0.14)	0.04	0.11	0.00	0.00
s	(0.02, 0.18)	(0.08, 0.73)	(0.00, 0.00)	(0.00, 0.00)	0.17	0.00	0.50	0.00
p1	(∞, ∞)	(∞, ∞)	(∞, ∞)	(∞, ∞)	0.27	0.00	0.00	0.00
sa	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)	(0.51, 0.81)	0.24	0.03	0.00	0.00
m1	(0.14, 0.17)	(0.00, 0.00)	(0.29, 1.00)	(0.00, 0.00)	0.02	0.00	0.00	0.61
m2	(0.50, 1.00)	(0.00, 0.00)	(1.00, 1.00)	(0.00, 0.00)	0.00	0.00	0.00	0.00

Table 7.14: Similarity with Group Profiles

Once user abnormal behavior is observed, we need to investigate the nature of the anomaly. Here we report our experiments on finding out how "user job functions" are violated. The problem can be stated as follows. Assume that there are n possible groups of users according to their job functions. When a user in group i does not behave according to the group profile (i.e., the normal job functions), we want to identify which group, e.g., group j, the user has become. That is, we want to know what "illegal" job functions the user has performed.

In our experiments, we first built group profiles for the p group (the programmers), the s group (the secretary), the m (the managers), and the sa group (the sysadm). From the data of the first 4 weeks, the patterns of all the users of a group were aggregated to form the group profile. The data of the 5th week was used to establish the range of similarity measures for all the users of each group. The user anomalies described in Table 7.12 include "illegal job function" cases during the 6th week: programmer1 becomes a secretary, secretary becomes a manager, and sysadm becomes a programmer. Table 7.14 compares the similarity measure of each user in an anomalous session with his/her normal similarity range gathered for the same time segment. From the normal similarity measures of each user with respect to the four groups, we can see that each user indeed has the largest similarity measure with his/her own group. From the similarity measures of the user anomalies, we can see that for each "illegal job function" case, the similarity measure of the targeted group is the largest. For example, when sysadm becomes a programmer, the similarity measure with the p group is the largest, and the similarity measure with the sa group is outside the normal range.

In summary, although formal evaluation statistics are not available to determine the error rates of our approach in user anomaly detection, the initial results are encouraging. We believe that our approach is worthy of future study.

7.2 Experiments on *BSM* Data

The DARPA data also contains Solaris BSM (Basic Security Module) [SunSoft, 1995] audit data for a designated host, *pascal*. In this section, we describe our experiments in building host-based intrusion detection models using BSM data.

When BSM is enabled in a host machine, there exists an *audit trail* for the host. An audit trail is a time-ordered sequence of actions that are audited on the system, and consists of one or more *audit files*. Each *audit record* in an audit file describes a single *audit event*, which can be a kernel event, i.e., a system call, or a user-level event, i.e., a system program (e.g., *inetd*, *in.rshd*, etc.) invocation.

We define *audit session* here as the collection of all audit events of an "incoming" or "outgoing" session on the host. Examples of these host sessions include *login* (e.g., terminal login, *telnet* login, *rlogin*, etc.), *rsh*, *ftp*, and *sendmail*, etc. It is easy to see that each host session often corresponds to a network connection. We can therefore correlate the predictions on host sessions by a host-based intrusion detection model with the predictions on the corresponding connections by a network intrusion detection model, to yield a higher accuracy.

As in the case of building network intrusion detection models, we also need to first perform a sequence of data preprocessing tasks on the raw BSM data. We extended the preprocessor component of USTAT [Ilgun, 1992] to process the binary BSM data into ASCII event data. Table 7.15 shows examples of the event records. Here a "?" means the value is not given in the original BSM audit record. Each event record contains a number of basic features, defined in Table 7.16.

We developed a program to process the event data into session records. A brief description of the procedure is the following:

time	auid	sid	event	pid	obname	...	ruid	euid
08:05:22	0	0	inetd_connect	0	?	...	0	0
...	...	...	...	...	...	...	...	...
08:05:22	-2	0	execve	415	/usr/sbin/in.telnetd	...	0	0
...	...	...	...	...	...	...	...	...
08:05:31	2104	417	setaudit	417	?	...	0	0
...	...	...	...	...	...	...	...	...
08:05:31	2104	417	chdir	418	/home/tristank	...	2104	2104
...	...	...	...	...	...	...	...	...

Table 7.15: Example BSM Event Records

- Watch for the beginning of a session, which is the execution of
  - the *inetd_connect* event (for *telnet*, *rlogin*, *rsh*, etc.), or
  - the *execve* event on a system program *in.fingerd* (for incoming *finger* request) or *finger* (outgoing), *mail.local* (incoming) or *sendmail* (outgoing), *ftpd* (incoming) or *ftp* (outgoing), etc.
- Record the *setaudit* event, which assigns the *auid* (audit user id) and *sid* (audit session id) of a session.
- Examine all audit records that share the same combination of *auid* and *sid* to summarize a number of session features, which are described in Section 7.2.1.
- Records the termination of a session.

The DARPA BSM data contains for each day about 500 sessions on host *pascal*. The vast majority of the intrusions in the BSM data are U2R buffer overflow attacks.

feature	**description**	**value type**
time	timestamp of the event	discrete
auid	audit user id, inherited by all child processes started by the user's initial process of a session	discrete
sid	audit session id, assigned for each login session and inherited by all descendant processes	discrete
event	audit event name	discrete
pid	process id of the event	discrete
obname	the name of the object, i.e., full path of the file, that the event operates on	discrete
arg1 - arg4	arguments of the system call	discrete
text	short information of the event, e.g., "successful login"	discrete
error_status	error status of the event	discrete
return_value	return value of a system call event	discrete
tmid	terminal id (port and ip address) of the event	discrete
ip header	the source and destination ip addresses and ports for the network connection handled by the event	discrete
socket	the local and remote ip addresses and ports of the socket used by the event	discrete
ruid	the real user id of the event	discrete
rgid	the real group id of the event	discrete
euid	the effective user id of the event	discrete
egid	the effective group id of the event	discrete

Table 7.16: Features of BSM Event Records

7.2.1 Defining Session Features

We experimented with feature construction for session records. We first computed frequent patterns from the event records. Here each dataset prepared for pattern mining contains all event records of a session that have positive (i.e., meaningful) *auid* and *sid*. That is, we are only interested in events that are "accountable" to the users.

Since we are looking for general rather than session-specific event patterns, we first removed *auid* and *sid* from the datasets. We also replaced *ruid* and *euid* with a flag *same_reid* to indicate whether *ruid* agrees with *euid*. We designated *event* as the *axis* attribute since it is obviously the most essential attribute in describing event data. We also used a *relative support* of 0.1 on both *event* and *obname* so that we can compute frequent patterns relative to the number of occurrences of each unique *event* and *obname*. For example, although an *obname* of */usr/bin/ksh* may occur only once or twice in the dataset of a session, we can still capture its patterns because using the relative support of 0.1, these occurrences are all "frequent".

After the initial few rounds of experiments, we discovered that the patterns are all related to very specific *obname* or *event* values. There are many kernel events (system calls) that cannot be directly linked to user-level commands. We reasoned that for intrusion detection purposes, we only need to analyze user-level commands and their operations on the file system. We therefore only kept the following types of event records: read, write, create, delete, execute, change owner or permission, rename, and link. The *event* value of each event record is replaced by the appropriate type name, e.g., *open_r* is replaced by *read*. We only kept the original *obname* if the event is *execute*, otherwise we used "user" to replace all *obname* values that indicate files in the user's directories, and "system" to replace the *obname* values that indicate files in the system's directories. We also removed all event records that have "?" (i.e., missing) *obname* values.

We aggregated event patterns of all normal sessions into a normal pattern set. And for each U2R session, we mined its event patterns and compared them with the normal patterns. When encoding the patterns, we used the following "order of importance" on the attributes: *same_reid*, *event*, *obname*, and the rest in alphabetical order. Applying the pattern encoding and comparison procedures discussed in Sections 6.2.1 and 6.3.1, we got the top 20% of intrusion-only patterns for each U2R attack, for example,

$$(event = execute, obname = /home/tristank/ffbexploit, same\_reid = 1),$$
$$(event = execute, obname = /usr/sbin/ffbconfig, same\_reid = 0) \rightarrow$$
$$(event = execute, obname = /usr/bin/ksh, same\_reid = 0)$$

and

$$(event = execute, obname = /usr/bin/pwd, same\_reid = 0) \rightarrow$$
$$(event = read, obname = home, same\_reid = 0)$$

These patterns are very "unique" because in the normal pattern set, patterns with $same\_reid = 0$ are those related to *read*, for example,

$$(event = read, obname = system, same\_reid = 0),$$
$$(event = read, obname = system, same\_reid = 0) \rightarrow$$
$$(event = read, obname = sytem, same\_reid = 0)$$

and

$$(event = read, obname = home, same\_reid = 0) \rightarrow$$
$$(event = execute, obname = /usr/bin/cat, same\_reid = 1)$$

We could have used a mechanical (automatic) pattern parsing and feature construction procedure for the intrusion-only patterns of the above forms. For example, we can add features that record the executions of the relevant specific events as described by the patterns. However, we very quickly realized that many U2R buffer overflow attacks share the same characteristics in their intrusion-only patterns. For example, another attack has the following intrusion-only pattern:

$$(event = execute, obname = /home/tristank/formatexploit, same\_reid = 1),$$
$$(event = execute, obname = /usr/bin/fdformat, same\_reid = 0) \rightarrow$$
$$(event = execute, obname = /usr/bin/ksh, same\_reid = 0)$$

These buffer overflow patterns indicate that there is an execution of a user program, follow by a SUID (setuid) system utility, and finally a shell in SUID state. We need to construct features that capture the general behavior of the attack method and the final outcome, rather than the specific system utilities being exploited. However, since the event data contains very low level and specific information of the operating system, we need to use domain knowledge to interprete the patterns to construct the more abstract and general features. Although our experiments here showed the limitations of (fully) automatic feature construction when dealing with low level event data, we believe that the intrusion-only patterns resulted from pattern mining and comparison can still provide a very helpful starting point for the manual feature definition process.

feature	**description**	**value type**
duration	length (number of seconds) of the session	continuous
service	operating system or network service, e.g., *telnet*, responsible for this session	discrete
logged_in	whether the user successfully logged in (when using *telnet*, rsh, etc).	discrete
failed_logins	number of failed login attempts	continuous
process_count	number of processes in the session	continuous
suid_sh	whether a shell is executed in suid state	discrete
suid_p	whether a suid system program is executed	discrete
user_p	whether a user program is executed	discrete
su_attempted	whether a su command is issued	discrete
access_files	number of write, delete, and create operations on access control files	continuous
file_creations	number of file creations	continuous
hot_login	whether the login belongs to the “hot” list	discrete
guest_login	whether the login belongs to the “guest” list	discrete

Table 7.17: Features of BSM Session Records

We defined a set of features, described in Table 7.17, for the BSM session records. Some of the features, i.e., those in bold, are from the buffer overflow patterns, while others are similar to the "content" features described in Session 7.1.2.

7.2.2 Misuse Detection Models on *BSM* Data

We labeled each BSM session record as "normal" or an intrusion name using the DARPA supplied list files. BSM session records from all 7 weeks were aggregated into a single dataset for training a misuse detection model. Table 7.18 shows some examples of the labeled BSM session records.

label	service	suid_sh	suid_p	user_p	file_creations	...
normal	smtp	0	0	0	0	...
normal	telnet	0	1	1	3	...
normal	telnet	0	1	0	0	...
buffer_overflow	telnet	1	1	1	2	...
normal	ftp	0	0	0	0	...
wraz_master	ftp	0	0	0	42	...
...	...	...	...	...	...	...

Table 7.18: Example BSM Session Records

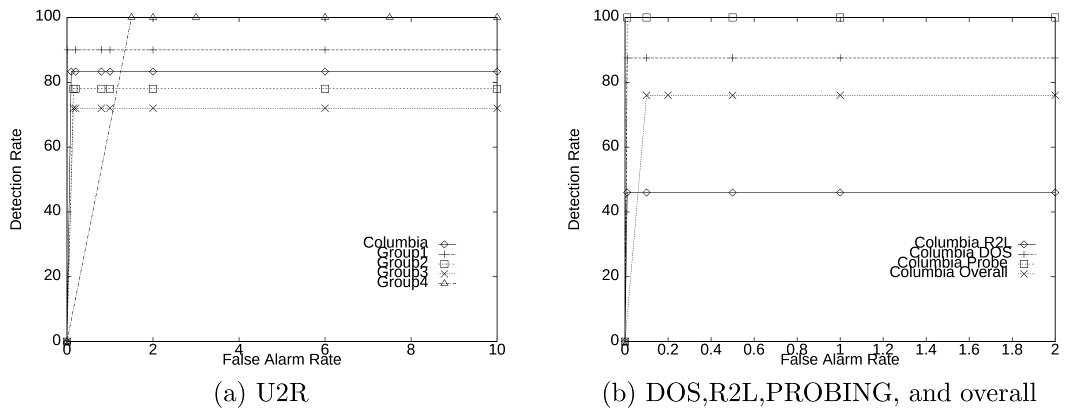
RIPPER rule	Meaning
buffer_overflow :- suid_sh = 1.	If a shell is executed in the SUID state, then this is a buffer overflow attack.
wraz_master :- file_creations $\geq$ 40, service = ftp, guest_login = 1.	If the service is ftp, the user logs in as a guest (*anonymous*), and the number of file creation operations is at least 42, then this is a wraz attack (in which the attacker logs to anonymous FTP site, creates a hidden directory, and uploads a lot of files, often pirated copies of software, for others to download).

Table 7.19: Example RIPPER Rules for BSM Session Records

RIPPER was applied to the dataset to learn a set of classification rules.

Table 7.19 shows some examples of these detection rules.

We evaluated the performance of the rule set on the test data using the list files provided by DARPA (available after the completion of the official evaluation). Figure 7.2 shows the ROC curves of the detection models on BSM data. Here sub-figure (a) compares our model with other participants in the DARPA Evaluation, in terms of performance in detecting the U2R attacks (DARPA only evaluated performance on U2R). Sub-figure (b) shows the performance of our model in detecting DOS, R2L, and PROBING attacks, as well as the overall performance (when all attacks are considered).



(a) U2R (b) DOS,R2L,PROBING, and overall

Figure 7.2: Performance of BSM Misuse Detection Models: ROC Curves on Detection Rates and False Alarm Rates

The model here has slightly better performance than the *tcpdump* models mainly because there are much fewer attacks contained in the BSM data (for the single host). We should note that the BSM model, like the *tcpdump* models, has good performance in detecting PROBING, U2R, and DOS attacks, but very poor performance in detecting R2L attacks. In fact, when we compared the predictions made by the BSM model and the *tcpdump* models, we found that they simply agree with each other's predictions (on the pairs of corresponding host session and network connection). That is, the BSM misuse detection model, as we have constructed, provides no additional predictive power. This is because:

- by the nature of misuse detection, a truly innovated intrusion, e.g., one that uses a service that was not modeled, can go undetected; and
- the features used in the BSM model and the *tcpdump* models are very similar, that is, the models are looking for similar set of evidences, although in different data sources.

While this seems discouraging if we have hoped for an accuracy improvement by combining the two models, this is in fact encouraging if our goal is to combine a light-weight *tcpdump* model that only checks the IP headers with a number of host-based models (for "important" hosts) that monitor the operating system activities. Our experiments here showed that the same level of accuracy can be maintained as using a heavy-weight *tcpdump* model that also checks the IP data contents.

7.3 Summary

In this chapter we described our experience in applying MADAM ID to build intrusion detection models on data provided by the 1998 DARPA Intrusion Detection Evaluation Program. We conducted two sets of experiments, one on network *tcpdump* data, and one on operating system BSM audit data. These experiments showed that we need to combine knowledge discovery (i.e., data mining) with knowledge engineering (i.e., domain knowledge encoding) to build effective intrusion detection models.

The experiments on *tcpdump* data showed the effectiveness of MADAM ID's automatic pattern mining and comparison, and feature construction procedures. We were able to compute a set of intrusion-only frequent patterns of DOS and PROBING attacks. These patterns were mechanically parsed to construct a set of temporal and statistical "traffic" features for the detection models. There are no intrusion-only patterns from connection records of R2L and U2R attacks because they normally involve a single connection. We used domain knowledge to define a set of "content" features for these attacks. An alternative approach would be to attempt mining patterns of the data contents of IP packets (and then pattern comparison and feature construction). This is not very attractive because the data contents are in free text, rather than structured, therefore we would still need domain knowledge to preprocess the contents to a suitable format, and to interprete the resultant patterns.

In the experiments on BSM data, we found that the intrusion-only patterns of buffer overflow attacks contain specific program names that are not necessarily inherent to the attack method. This is because, compared with connection records (with the set of "intrinsic" features) which are more general and their semantics well understood, the BSM audit event records are about low-level details of system activities. We therefore need to use domain knowledge to interprete these patterns and define features so that only the most general information is summarized into BSM session records.

The official DARPA Evaluation results showed that our detection models on *tcpdump* data were one of the best among all participants. This shows the advantages of using MADAM ID to process the huge amount of audit data, construct features, and inductively learn classification rules.

However, since our models were intended for misuse detection, and were trained using only the available training data, a number of new attacks in the test were not detected. We have just begun research in very challenging problem of building anomaly detection models.

Chapter 8

Towards Rapid Deployment of Real-time Intrusion Detection Systems: Issues and Initial Results

In this chapter, we describe our on-going project in real-time network intrusion detection. We first discuss our approach for automatically translating the RIPPER rules into NFR (Network Flight Recorder) real-time N-code modules. We then outline the key issue involved in real-time intrusion detection: how to efficiently execute the detection models so that intrusions are detected with minimum delay and minimum cost of computation.

Our solution is to mine the "necessary conditions" that consist of low-cost and short-delay features for each intrusion. These necessary conditions can be used to filtered out a large number of unnecessary feature computation and rule checking. An orthogonal solution is to directly learn RIPPER rules that are sensitive to timing cost of the features.

8.1 Automatic Translation of RIPPER Rules into Real-time NFR Modules

Our intrusion detection models are produced off-line. Effective intrusion detection should be in real-time to minimize security compromises. We therefore need to study how our models preform in a real-time environment. We are working on translating RIPPER rules into real-time detection modules in NFR (Network Flight Recorder) [Network Flight Recorder Inc., 1997], a system that includes a packet capturing engine and N-code programming support for specifying packet "filtering" logic.

NFR offers a fairly simple framework for network monitoring. It sniffs packets from the network, reassembles them, and then passes them to filter functions for further processing, e.g., calculating network traffic statistics. Filter functions are written in N-code, a preemptive, event-driven scripting language that supports a wide range of operations on network packet data. In order to use NFR for network intrusion detection, one needs to implement an extensive set of N-code filters that look for evidences in the network packets.

The purposes of our NFR project are therefore not just to simply evaluate how off-line learned models can be utilized in a real-time environment, but more importantly, to automate the creation of N-code real-time detection filters.

A detection rule, for example,

```
pod :- wrong_fragment >= 1, protocol_type = icmp.
```

can be automatically converted into the following N-code:

```
filter pod () {
   if (wrong_fragment() > 1 && protocol_type () == icmp)
       alarm_pod ();
}
```

as long as the features, i.e., wrong_fragment and protocol_type, have been implemented as N-code filter functions.

Our plan is to implement all the required features used in the RIPPER rule set as N-code "feature filters", and implement a translator that can automatically translate each RIPPER rule into an N-code "rule filter". We envision that when NFR is shipped with MADAM ID to a customer site, MADAM ID can be used to learn the site-specific intrusion detection rules using the locally gathered audit data, and be automatically converted into N-code "rule filters". We believe that our NFR project will showcase how to rapidly and automatically deploy and customize an IDS.

8.2 Efficient Execution of Learned Rules

Although often ignored in off-line analysis, efficiency is a very important consideration in real-time intrusion detection. In our first experimental implementation of N-code "rule filters", we essentially tried to follow the off-line analysis steps in a real-time environment. A connection is not inspected (i.e., classified using the rules) until its connection record is completely formulated, that is, all packets of the connection have arrived and summarized, and all the temporal and statistical features are computed. This scheme failed miserably. When there is a large volume of network traffic, the amount of time taken to process the connection records within the past 2 seconds and calculate the statistics is also very large. Many connections may have terminated (and thus completed with attack actions) when the current connection is finally inspected by the RIPPER rules. That is, the detection of intrusions is severely delayed. Ironically, DOS attacks, which typically generate a large amount a traffic in a very short period time, are often used by intruders to first overload an IDS, and use the detection delay as a window of opportunity to quickly perform their malicious intent. For example, they can seize control of the operating system and "kill" the IDS.

We need to examine the time delay associated with each feature in order to speed up the model execution. The time delay of a feature includes not only the time of its computation, but also the time of its readiness (i.e., when it can be computed). For example, the *flag* of a connection can only be computed (summarized) after the last packet of the connection has arrived, whereas the *service* of a connection can be obtained by checking the header of the first packet.

We partition the features into 3 "cost" (time delay) levels: level 1 features can be computed from the first packet; level 2 features can be computed at the end of the connection, using only information of the current connection; level 3 can be computed at the end of the connection, but require access to data of (many) other prior connections. As a datum arrives early in the "data flow", shown in Figure 3.1, the cost will be lower to calculate the feature that depends upon that datum. In order to conveniently estimate the cost of a rule, we assign a cost of 1 to the level 1 features , 10 to level 2, and 100 to level 3. That is, the different levels have an order of magnitude difference in cost. For the feature set derived from the DARPA dataset, *service* is a level 1 feature, all the other “intrinsic” and “content” features are in level 2, and all “traffic” features are in level 3.

Note that we cannot simply order the rules by their costs for real-time execution for the following reasons. First, the rules output by RIPPER are in a strict sequential order (e.g., “if rule 1 else rule 2 else ...”), and hence reordering the rules may result in unintended classification errors. Furthermore, even if the rules can be tested in strictly cost order without introducing classification error, many rules will still be tested (and fail to match) before a classification is made. That is, ordering the rules by their costs alone is not necessarily the optimal solution for quick model evaluation. We thus seek to compute an “optimal schedule” for feature computation and rule testing to minimize model evaluation costs, and increase the response rate for real-time detection.

8.2.1 Low Cost “Necessary” Conditions

Ideally, we can have a few tests involving the low cost (i.e., level 1 and level 2) features to eliminate the majority of the rules that need to be checked, and thus eliminating the needs to compute some of the high cost features.

In order to eliminate the need for testing a rule for intrusion I, we need a test of the form of

$$F \to \neg I$$

which can be derived from

$$I \rightarrow \neg F$$

We can compute the association rules that have the intrusion labels on the LHS and the low cost features on the RHS, and a *confidence* of 100%.

We discovered several such associations for the RIPPER rules, for example,

$$\begin{aligned}
&\textit{ping\_of\_death} \rightarrow \textit{service} = \textit{icmp\_echo}\ [c = 1.0]\\
&\textit{phf} \rightarrow \textit{service} = \textit{http}\ [c = 1.0],\\
&\textit{port\_scan} \rightarrow \textit{src\_bytes} = 0\ [c = 1.0],\ \text{and}\\
&\textit{syn\_flood} \rightarrow \textit{flag} = \textit{S0}\ [c = 1.0],\ \text{etc.}
\end{aligned}$$

Note that most of these feature values, for example, $\textit{src\_bytes} = 0$, are not in the RIPPER rules because they are prevalent in the normal data. That is, they don't have predictive power. However, these associations are the "necessary" conditions for the intrusions, for example, "this connection is a port-scan attack **only if** *src_bytes* is 0", which is equivalent to "if the *src_bytes* is **not** 0, then this connection is **not** a port-scan attack".

Note that when the RHS of such associations has n feature value pairs (regarding to different features), there are a corresponding n independent necessary conditions. We can always select the one with the lowest cost. We can also merge associations,

$$\begin{aligned}
&I \rightarrow A_i = v_1\ [c_1]\\
&I \rightarrow A_i = v_2\ [c_2]\\
&\ldots\\
&I \rightarrow A_i = v_n\ [c_n]
\end{aligned}$$

where $\sum_{i=1}^{n} c_i = 1.0$, into a single association,

$$I \rightarrow A_i = v_1 \vee v_2 \ldots \vee v_n \; [c = 1.0]$$

For example, we have from the DARPA data,

$$\mathit{buffer\_overflow} \rightarrow \mathit{service} = \mathit{telnet} \; [c = 0.91]$$

and

$$\mathit{buffer\_overflow} \rightarrow \mathit{service} = \mathit{rlogin} \; [c = 0.09]$$

which are merged to

$$\mathit{buffer\_overflow} \rightarrow \mathit{service} = \mathit{telnet} \vee \mathit{rlogin} \; [c = 1.0]$$

When a RIPPER rule for an intrusion is excluded because of the failure of its necessary condition, the features of the rule need not be computed, unless they are needed for other candidate (remaining) rules. We next discuss how to do efficient bookkeeping on the candidate rules and features to determine a schedule for feature computation and rule condition testing.

8.2.2 Real-time Rule Filtering

Suppose that we have n RIPPER rules. We use a n-bit vector, with the bit order corresponding to the order of the rules output by RIPPER, as the "remaining" vector to indicate which rules still need to be checked. Initially, all bits are 1's. Each rule has a "invalidating" n-bit vector, where only the bit corresponding to the rule is 0 and all other bits are 1's. Each of the high cost features, i.e., the level

3 temporal and statistical feature, has a "computing" n-bit vector, where only the bits corresponding to the rules that require this feature are 1's.

For each intrusion label, we record its "lowest cost necessary condition" (if there are such conditions), according to the costs of the features involved. We sort all these necessary conditions according to the costs to produce the order for real-time condition checking.

When examining a packet, or a (just completed) connection, if a "necessary" condition of an intrusion is violated, the corresponding "invalidating" bit vectors of the RIPPER rules of the intrusion are used to AND the "remaining" vector and all the "computing" vectors for the high cost features. After all the necessary conditions are checked, we get all the features with non-zero "computing" vectors. These features are potentially useful because of the remaining rules that need to be checked. A single function call is made to N-code modules to compute all these features at once. This execution strategy reduces memory or disk access since these features compute statistical information on the past (stored) connections records. The "remaining" vector is then used to check the remaining rules one by one.

We are currently fine tuning our implementation of this scheme and need to perform an extensive set of experiments, simulating a wide variety of intrusions, to establish the empirical speed-up we may attain. However, our analysis on the necessary conditions for DOS and PROBING attacks, and the set of features used by their RIPPER rules, suggest that one or two simple low cost tests (e.g., *service* and/or *flag*) can reduce the number of high cost feature tests from 9 (see Table 7.8) to at most 3. Our preliminary experiments have thus far confirmed this result.

8.3 An Orthogonal Approach: Constructing Cost-Sensitive Rules

Rather than first learning a set of rules that consider only accuracy, and then adding cost consideration in rule execution, we can begin with learned rules that are sensitive to the computational costs of the features.

The simplest way of incorporating attribute costs into the process of learning a classification model is to modify the search heuristic, e.g., information gain, to also consider cost. For example, following the discussion in [Mitchell, 1997], let *Gain*(A) be the FOIL information gain in RIPPER that evaluates a conjunct involving attribute A, and let *Cost*(A) be the cost of computing (obtaining) a value of A, we can use

$$\frac{2^{Gain(A)} - 1}{(Cost(A) + 1)^w}$$

as the evaluation function. Here $w \in [0, 1]$ is a constant that determines how sensitive, i.e., what is the relative importance, of cost versus of information gain. Using such modified evaluation function, RIPPER essentially considers both accuracy and attribute costs when constructing classification rules.

Note that this approach is orthogonal to the previous approach because although the overall cost of each rule may be lower, we still need to optimize the execution schedule of the rules.

8.4 Current Status and Near-term Plans

Currently, we are near completion in implementing all the required features as N-code functions. Implementation of the automatic rule-to-N-code translator is well under way.

We have modified RIPPER to consider feature costs along with information gain when constructing classification rules. We have implemented mechanisms to measure the actual real-time feature costs, in terms of computation times of the corresponding N-code functions.

In the next three months, we hope to complete the following tasks:

- implementation of the automatic translator as well as all the features;
- extensive experiments to obtain estimates of the real-time feature costs, which will be parameterized into RIPPER to learn a set of cost-sensitive detection rules;
- implementation of the real-time rule filtering scheme;
- evaluation of the two approaches, using both accuracy and computation cost.

8.5 Summary

In this chapter we discussed the key advantages of incorporating off-line learned rules into a real-time intrusion detection system. By implementing the needed features as real-time inspection modules, a classification rule can be straightforwardly translated into a sequence of function calls to the feature modules. Therefore, deploying and customizing an IDS, which is currently an enormous pure knowledge engineering task, can be accomplished by applying data mining programs to learn the local detection rules from the local audit data, and automatically translating the rules into detection modules of the IDS. We believe this approach will significantly facilitate the widespread deployment of well-engineered IDSs.

We discussed the efficiency issue in real-time intrusion detection. We proposed two orthogonal approaches. In the first approach, we can compute a set of low-cost "necessary" conditions for each rule. These conditions can be used to dynamically filter out "unnecessary" high-cost feature computation and rule checking. In the second approach, we can extend the information gain measure to also consider feature costs so that the resultant rules are sensitive to cost.

We are in the initial stage of a pilot project with NFR (Network Flight Recorder), a real-time network IDS.

Chapter 9

Conclusions

In this chapter, we summarize the thesis, review the thesis contributions, and discuss future work.

9.1 Summary

This thesis described a framework, MADAM ID, for Mining Audit Data for Automated Models for Intrusion Detection. MADAM ID consists of classification and meta-classification programs, association rules and frequent episodes programs, and a feature construction system. Using MADAM ID, frequent patterns computed from large amount of system audit data are used to construct predictive features, from which classification rules are inductively learned to detect intrusions.

We first motivated our thesis by stating the importance of intrusion detection in the overall computer security mechanisms. We provided background on intrusion detection techniques, and briefly described representative intrusion detection systems. We pointed out that current intrusion detection systems lack effectiveness, adaptability, and extensibility because of the pure knowledge engineering development approaches. The goal of this thesis research is therefore to develop a framework that facilitates automatic and systematic construction of adaptable and extensible intrusion detection systems. We explained that the knowledge discovery (data mining) process involves many different algorithms and tools. For the purposes of building intrusion detection models, we need classification programs, and patterns mining programs, namely, the association rules and frequent episodes algorithms.

We examined the problem of learning classification models. We discussed that for building classification models from audit data, the most important issue is to construct a set of features that are likely to have high information gain measures. Therefore the main focus of this thesis research is on feature construction from audit data. Through the experiments on building classification models using *sendmail* system call data and network *tcpdump* data, we demonstrated that classification rules can be accurate, concise, and intuitive. These experiments also showed that we need to gather sufficient training data and construct a set of predictive features. Both of these tasks require proper data analysis tools. We proposed to use frequent patterns from audit data, which are statistically summaries of system activities, as guidelines for audit data gathering and feature construction.

We then studied the problem of mining frequent patterns from audit data. Given the temporal and multi-attribute nature of system audit data, we argued that two kinds of frequent patterns need to be computed: association rules, which describe frequent correlations among attributes, and frequent episodes, which capture frequent co-occurrences of system events. The basic association rules and frequent episodes algorithms are exponential in the worst-cases. To efficiently compute only the "useful" patterns from the large amounts of audit data, we extended the basic algorithms to utilize the "schema-level" information about audit data. These extensions can be summarized as "measuring the interestingness of a pattern by the attributes involved, in addition to its frequency values". The specific extensions include the use of *axis* attribute(s) to find patterns related to the most important attribute(s), and to unify associations and frequent episodes into a single rule formalism; the use of *reference* attribute to compute the "same-subject" frequent patterns; and the use of *level-wise* approximate mining and *relative* support to uncover low frequency but important patterns.

We next studied the problem of how to utilize the mined frequent patterns. We can aggregate patterns from each audit dataset into a pattern set, where two similar rules are *merged* into one. Our experiments showed that the stabilization of the pattern set can be used as an indicator that sufficient audit data has been gathered. In order to efficiently analyze the large amounts of frequent patterns, we devised a simple encoding scheme which, according to the user-specified "order of importance" among attributes, maps patterns that are syntactically and structurally more similar into closer numbers. Our visualization experiments showed that this encoding scheme facilitates systematic navigation and analysis, e.g., *roll-up*, *drill-down*, and *segmentation* of the pattern space. These experiments also showed that the encoded normal patterns and intrusion patterns form distinct "clusters". We then devised a pattern comparison procedure based on the pattern encodings. When compared with the aggregated normal pattern set, the patterns from an intrusion dataset that have the user-specified top percent of *different* scores are identified as the “intrusion-only” patterns. We designed an automatic feature construction algorithm that parses an intrusion-only pattern, and applies *count*, *percent*, and *average* operators on the occurrences of attribute values to add temporal and statistical features. We explained the theoretical underpinnings to show that the constructed features will have high information gain because their values distinguish intrusion records from normal records.

We next described the objective evaluation of the algorithms and tools of MADAM ID. In the 1998 DARPA Intrusion Detection Evaluation Program, we applied MADAM ID to build network intrusion models on *tcpdump* data. We showed that we need to combine knowledge discovery with knowledge engineering to build effective intrusion detection models. For DOS and PROBING attacks, we were able to automatically construct a set of temporal and statistical “traffic” features using the frequent patterns mined from the network connection records. However, for U2R and R2L attacks, we had to rely on domain knowledge to define the “content” features because mining patterns from unstructured data contents is extremely difficult. Similarly, when building models using BSM data, we needed to use domain knowledge to interprete and abstract the often too specific intrusion-only patterns when defining host session features. Although the DARPA evaluation showed that our misuse detection models had one of the best performances among all participants, they are ineffective in detecting truly innovative attacks. Therefore, we need to study how to build anomaly detection models.

We are currently studying the problem of how to incorporate our off-line learned models into real-time IDSs, in a pilot project with NFR (Network Flight Recorder). The main idea is to implement all the required features as N-code filter functions, and automatically translate each learned rule in to an N-code filter, which is essentially a sequence of function calls to the implemented feature filters. Deployment and customization of NFR will become very easy because the new site-specific detection rules can be learned by applying MADAM ID to the local audit data, and then be translated into NFR. The main challenge is how to efficiently execute the rules in a real-time environment. One approach is to use low-cost "necessary" conditions for each rule to filter out "unnecessary" feature computation and rule checking in real-time. An orthogonal approach is to extend the information gain measure used by the machine learning program, e.g., RIPPER, to consider the feature costs so that the resultant rules are sensitive to cost.

9.2 Thesis Contributions

We recap the thesis contributions:

- **Extentions to the Association Rules and Frequent Episodes Algorithms** We studied the problem of how to efficiently compute only the "relevant" frequent patterns from the large amounts of audit data. We exploited the schema-level information of audit data. We extended the basic algorithms to use *axis* attribute(s) and *reference* attribute(s) as forms item constraints; and to use *level-wise* approximate mining and *relative* support to compute the low frequency but important features.

- **Techniques for Pattern Visualization and Comparisons** We studied the problem of how to efficiently analyze and compare frequent patterns. We devised an encoding scheme that maps similar patterns to closer numbers, and a pattern comparison procedure that is based on the *difference* scores of pattern encodings. We applied the patten encoding and comparison methods to identify the "intrusion-only" patterns.

- **Techniques for Automatic Feature Construction from Patterns** We studied the problem of how to automatically construct features from the mined patterns. We designed an algorithm that parses a frequent pattern and adds temporal and statistical features. We have applied this algorithm to construct predictive features from "intrusion-only" patterns.

- **Techniques for Efficient Real-time Execution of Detection Models** We just began to study the problem of how to efficiently execute the off-line learned detection models in a real-time environment. We developed a technique of utilizing low-cost "necessary" conditions as a mechanism for filtering and scheduling rules in run-time.

- **Objective Evaluation of MADAM ID** We participated in the 1998 DARPA Intrusion Detection Evaluation Program. To the intrusion detection field, we showed the advantages of combining knowledge discovery with knowledge engineering. To the data mining field, we showed the strengths as well as limitations of current techniques and algorithms.

9.3 Future Work

There are several interesting and important future directions:

- **Anomaly Detection** We have thus far done very little study on anomaly detection, except using the *sendmail* system call data to construct a model of program normal execution, and using shell command data to construct user models. Anomaly detection in a broader scale, e.g., for a network, is very important and much more complex. The biggest challenge is how to control the false positive rate. Given the wide range normal behavior of the services and hosts in a network, it is very likely that a detected anomaly is actually normal (i.e., legitimate). A solution is to divide-and-conquer, that is, to create host groups and service groups and learn a dedicated normal profile for each group, and in addition, a normal profile on the predictions of these group models. This is in essence the spirit of hierarchical meta-learning.

- **Integrating IDSs with Network Management Systems** We believe that it is important, beneficial, and only natural to integrate an intrusion detection with a network management system. A lot of network anomalies can be filtered by a network management system first because this is part of its function. On the other hand, when detecting an intrusion, the IDS can communicate the network management system to take appropriate actions, e.g., re-route some services from a compromised host.

- **Cost-sensitive Intrusion Detection** There are many cost factors in intrusion detection, for example, the cost of computing features and checking a rule (which we are beginning to address), the cost of identifying an intrusion (the labor cost of system staff called upon to take actions), and the cost (the damage) of an intrusion, etc. These are practical issues that need to be considered when deploying an IDS. The research challenge here is to build intrusion detection models that can be easily adjustable according to a site-specific "cost policies".

- **Formal Design of Intrusion Detection Techniques** There is currently no formal reasoning behind intrusion detection techniques. We don't know *a priori* whether a method would work. And when an intrusion detection model fails, we can't prove that it is the fault of the modeling algorithm, the lack of "meaningful" data, or both. We are limited by the current state of design and implementation practices of operating systems and network services. There are no (or very little) formal specifications for our computing components and the interactions among them. Since we have no way of reasoning the possible behavior of a system, we can only *observe*, which involves a lot of imprecise guesswork. We need to address these problems by working with the operating system and networking communities. Although it is unlikely that a system component will be designed and implemented using formal methods in every single step, we can require that its test plans (i.e., what are its intended and well-tested usages) be released to the IDS community, along with a specification of its auditing mechanisms (what can or cannot be recorded, and from which data sources). We can then reason the best possible models for such a system component.

- **Predictive Models for Sequence Data** MADAM ID uses frequent patterns to construct temporal and statistical features to build classification models. It will be interesting to see whether this multi-step operation can be combined into a single-step. That is, can we mine predictive models from the time-sequenced audit records? Most prior work dealt with single-dimensional data streams. Multi-dimensional audit records presents a combinatorial problem, thus the challenge is to develop an efficient and scalable algorithm.

9.4 Closing Remarks

This thesis documented our research in developing and applying data mining techniques to the important and challenging problem of building intrusion detection models. A set of algorithms and tools, collectively called MADAM ID, have been designed, implemented, and evaluated for this thesis research. While MADAM ID has shown great promises, there are open problems for future research.

Bibliography

[Agrawal and Srikant, 1994] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994.

[Agrawal and Srikant, 1995] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, 1995.

[Agrawal *et al.*, 1993] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 207–216, 1993.

[Atkins *et al.*, 1996] D. Atkins, P. Buis, C. Hare, R. Kelley, C. Nachenberg, A. B. Nelson, P. Phillips, T. Ritchey, and W. Steen. *Internet Security Professional Reference*. New Riders Publishing, 1996.

[Bellovin, 1989] S. M. Bellovin. Security problems in the TCP/IP protocol suite. *Computer Communication Review*, 19(2):32–48, April 1989.

[Brachman *et al.*, 1996] R. J. Brachman, T. Khabaza, W. Kloesgen, G. Piatetsky-Shapiro, and E. Simoudis. Mining business databases. *Communications of the ACM*, 39(11):42–48, November 1996.

[Chan and Stolfo, 1993] P. K. Chan and S. J. Stolfo. Toward parallel and distributed learning by meta-learning. In *AAAI Workshop in Knowledge Discovery in Databases*, pages 227–240, 1993.

[Cheeseman and Stutz, 1996] P. Cheeseman and J. Stutz. Bayesian classification (autoclass): Theory and results. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press, Cambridge, MA, 1996.

[Clark and Niblett, 1989] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3(4):261–283, 1989.

[Cohen, 1994] W. W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.

[Cohen, 1995] W. W. Cohen. Fast effective rule induction. In *Machine Learning: the 12th International Conference*, Lake Taho, CA, 1995. Morgan Kaufmann.

[Crosbie and Spafford, 1995] M. Crosbie and E. H. Spafford. Active defense of a computer system using autonomous agents. Technical Report CSD-TR-95-008, COAST Laboratory, Department of Computer Science, Purdue University, West Lafayette, IN, 1995.

[Domingos, 1998] P. Domingos. Occam's two razors: The sharp and the blunt. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*, New York, NY, August 1998. AAAI Press.

[Faloutsos and Lin, 1995] C. Faloutsos and K. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, San Jose, CA, May 1995.

[Fawcett and Provost, 1996] T. Fawcett and F. Provost. Combining data mining and machine learning for effective user profiling. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, pages 8–13, Portland, OR, August 1996. AAAI Press.

[Fayyad *et al.*, 1996a] U. Fayyad, D. Haussler, and P. Stolorz. Mining scientific data. *Communications of the ACM*, 39(11):51–57, November 1996.

[Fayyad *et al.*, 1996b] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. From data mining to knowledge discovery: An overview. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*. AAAI/MIT Press, Cambridge, MA, 1996.

[Fayyad *et al.*, 1996c] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process of extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, November 1996.

[Forrest *et al.*, 1996] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on*

Security and Privacy, pages 120–128, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[Frank, 1994] J. Frank. Artificial intelligence and intrusion detection: Current and future directions. In *Proceedings of the 17th National Computer Security Conference*, October 1994.

[Grampp and Morris, 1984] F. T. Grampp and R. H. Morris. Unix system security. *AT&T Bell Laboratories Technical Journal*, 63(8):1649–1672, October 1984.

[Han and Fu, 1995] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proceedings of the 21th VLDB Conference*, Zurich, Switzerland, 1995.

[Heady *et al.*, 1990] R. Heady, G. Luger, A. Maccabe, and M. Servilla. The architecture of a network level intrusion detection system. Technical report, Computer Science Department, University of New Mexico, August 1990.

[Hătőnen *et al.*, 1996] K. Hătőnen, M. Klemettinen, H. Mannila, and P. Ronkainenand H. Toivonen. TASA: Telecommunication alarm sequence analyzer. In *Proceedings of the IEEE/IFIP 1996 Network Operations and Management Symposium*, April 1996.

[Ilgun *et al.*, 1995] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[Ilgun, 1992] Koral Ilgun. USTAT: A real-time intrusion detection system for Unix. Master's thesis, University of California at Santa Barbara, November 1992.

[Jacobson *et al.*, 1989] V. Jacobson, C. Leres, and S. McCanne. *tcpdump*. available via anonymous ftp to ftp.ee.lbl.gov, June 1989.

[Jonsson and Olovsson, 1997] E. Jonsson and T. Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Transactions on Software Engineering*, 23(4), April 1997.

[Klemettinen *et al.*, 1994] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 401–407, Gainthersburg, MD, 1994.

[Ko *et al.*, 1994] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, December 1994.

[Kumar and Spafford, 1995] S. Kumar and E. H. Spafford. A software architecture to support misuse intrusion detection. In *Proceedings of the 18th National Information Security Conference*, pages 194–204, 1995.

[Lee and Stolfo, 1998] W. Lee and S. J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[Lee *et al.*, 1998] W. Lee, S. J. Stolfo, and K. W. Mok. Mining audit data to build intrusion detection models. In *Proceedings of the 4th International Conference* *on Knowledge Discovery and Data Mining*, New York, NY, August 1998. AAAI Press.

[Lee *et al.*, 1999a] W. Lee, S. J. Stolfo, and K. W. Mok. A data mining framework for building intrusion detection models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, May 1999.

[Lee *et al.*, 1999b] W. Lee, S. J. Stolfo, and K. W. Mok. Mining in a data-flow environment: Experience in network intrusion detection. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD-99)*, August 1999.

[Lent *et al.*, 1997] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *Proceedings of the 13th International Conference on Data Engineering*, Birmingham, UK, 1997.

[Lunt *et al.*, 1992] T. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. Neumann, H. Javitz, A. Valdes, and T. Garvey. A real-time intrusion detection expert system (IDES) - final technical report. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, February 1992.

[Lunt, 1993] T. Lunt. Detecting intruders in computer systems. In *Proceedings of the 1993 Conference on Auditing and Computer Technology*, 1993.

[Mannila and Toivonen, 1996] H. Mannila and H. Toivonen. Discovering generalized episodes using minimal occurrences. In *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, Portland, Oregon, August 1996.

[Mannila *et al.*, 1995] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proceedings of the 1st International Conference on Knowledge Discovery in Databases and Data Mining*, Montreal, Canada, August 1995.

[McClure *et al.*, 1998] S. McClure, J. Scambray, and J. Broderick. Test Center Comparison: Network intrusion-detection solutions. In *INFOWORLD May 4, 1998*. INFOWORLD, 1998.

[Mitchell, 1997] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[Network Flight Recorder Inc., 1997] Network Flight Recorder Inc. Network flight recorder. http://www.nfr.com, 1997.

[Padmanabhan and Tuzhilin, 1998] B. Padmanabhan and A. Tuzhilin. A belief-driven method for discovering unexpected patterns. In *Proceedings of the 4th International Conference on Knowledge Discovery and Data Mining*, New York, NY, August 1998. AAAI Press.

[Paxson, 1997] V. Paxson. End-to-end internet packet dynamics. In *Proceedings of SIGCOMM '97*, September 1997.

[Paxson, 1998] V. Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998.

[Porras and Neumann, 1997] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *National Information Systems Security Conference*, Baltimore MD, October 1997.

[Porras and Valdes, 1998] P. A. Porras and A. Valdes. Live traffic analysis of TCP/IP gateways. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, March 1998.

[Quinlan, 1986] J. R. Quinlan. Induction on decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Quinlan, 1990] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Rumelhart *et al.*, 1994] D. Rumelhart, B. Widrow, and M. Lehr. The basic ideas in neural networks. *Communications of the ACM*, 37(3):87–92, 1994.

[Srikant *et al.*, 1997] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 67–73, Newport Beach, California, August 1997. AAAI Press.

[Stevens, 1994] W. R. Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley Publishing Company, 1994.

[Stolfo *et al.*, 1997a] S. J. Stolfo, W. Fan, W. Lee, A. L. Prodromidis, and P. K. Chan. Credit card fraud detection using meta-learning: Issues and initial results. In *AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*, pages 83–90. AAAI Press, July 1997.

[Stolfo *et al.*, 1997b] S. J. Stolfo, A. L. Prodromidis, S. Tselepis, W. Lee, D. W. Fan, and P. K. Chan. JAM: Java agents for meta-learning over distributed databases. In *Proceedings of the 3rd International Conference on Knowledge*

Discovery and Data Mining, pages 74–81, Newport Beach, CA, August 1997. AAAI Press.

[SunSoft, 1995] SunSoft. *SunSHIELD Basic Security Module Guide.* SunSoft, Mountain View, CA, 1995.

[Teng *et al.*, 1990] H. S. Teng, K. Chen, and S. C. Lu. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 278–284, Oakland CA, May 1990.

[Utgoff *et al.*, 1997] P. E. Utgoff, N. C. Berkman, and J. A. Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29:5–44, October 1997.