Incremental Bayesian Segmentation for Intrusion Detection

by

Joseph R. Hastings

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2003

Author . . . .
Department of Electrical Engineering and Computer Science
September 8, 2003

Certified by . . . .
Peter Szolovits
Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Incremental Bayesian Segmentation for Intrusion Detection

by

Joseph R. Hastings

Submitted to the Department of Electrical Engineering and Computer Science
on September 8, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes an attempt to monitor patterns of system calls generated by a Unix host in order to detect potential intrusion attacks. Sequences of system calls generated by privileged processes are analyzed using incremental Bayesian segmentation in order to detect anomalous activity. Theoretical analysis of various aspects of the algorithm and empirical analysis of performance on synthetic data sets are used to tune the algorithm for use as an Intrusion Detection System.

Thesis Supervisor: Peter Szolovits
Title: Professor

Acknowledgments

I would like to thank Professor Peter Szolovits for initially suggesting the use of IBS and for his advice and support during the past 18 months. I would like to thank Dr. Marco Ramoni for all of his help in explaining the algorithm, helping me to debug my enhancements, and suggesting interesting avenues of research. I would like to thank Professor Alan Edelman for his advice in implementing the MPI version of the code as well as helping me to prove several of the mathematical results. I would like to thank Professors Leslie Kaelbling, Tomas Lozano-Perez, and Tommi Jaakkola for their help in answering questions about machine learning. Finally I would like to thank Ben Richardson, Dave Milavich, Dave Wilson, Mark Rosen, Erica Peterson, Sid Sen, and Eric Jonas for all of their help and support throughout the course of the thesis. This research was supported in part by the Cyber Command and Control Program of the Defense Advanced Research Projects Agency under contract number F30602-99-1-0509.

Contents

List of Figures

Chapter 1

Introduction

Intrusion detection systems (IDS) have become an important part of security systems in use by institutions of all sizes. The concept and terminology of intrusion detection is widely attributed to J. Anderson [1] and the concept introduced in 1980 has grown into a multi-billion dollar industry. Due to the massive amounts of data available for monitoring the behavior of a complex computer system, the problem is well suited to computerized automation in conjunction with limited human interaction. Various machine learning techniques have been applied to intrusion detection in order to create classifiers that automatically distinguish between normal and abnormal system behavior. D. Denning is recognized as one of the pioneers of applying machine learning to intrusion detection in 1986 [8]. This thesis follows in that tradition and describes an attempt to apply *Incremental Bayesian Segmentation* (IBS), a relatively new machine learning algorithm, to the domain of intrusion detection. In particular, IBS is used to construct a real-time monitor that continually assesses the likelihood that a computer system is behaving abnormally.

This thesis is organized as follows: the remainder of the introduction describes general IDS, previous attempts to use system calls to identify attacks, the derivation of the IBS algorithm, and the construction of a real-time IDS using IBS. The implementation section discusses an IDS written using IBS. The experiments section describes various tests that were performed to tune the IBS algorithm for the particular domain of intrusion detection. Finally, the conclusion section discusses the advantages and disadvantages of IBS and its role as part of a hypothetical IDS suite.

1.1 Intrusion Detection Systems

For the purposes of this thesis, the term *intrusion* refers to the deliberate and successful altering of the behavior of a computer system such that it produces effects unintended by its owner.[1] This differs slightly from previously used definitions [1, 4, 9] as it simplifies the concept of an owner to a single entity that can perfectly characterize system behavior as normal or abnormal. Let the term *agent* refer to any entity that has the capability of altering the behavior of the computer system being considered. Agents may be in the form of human users, other computer systems, or even portions of the computer system itself. Define *users* to be agents whose intentions, for the relevant time-frame being considered, are consistent with those of the owner of the computer system. Agents that are not users are referred to informally as *attackers*.[2] The actions of attackers are known as *attacks* when they have the intent of altering the behavior of the computer system. Note that all of these definitions rely on the intentions of the agents involved and not on their actions or effects. For this reason these terms are difficult to interpret in any real-world scenario in which intentions themselves have no clear definition.[3] This thesis quickly shifts the focus away from the intentions of the agents to the actual response of the computer system.

In general, there are many types of agents and many degrees to which their intentions may disagree with those of the owner of a computer system. Any particular security system identifies a *threat model* stating which types of actions it is interested in detecting. For most IDS, the threat model is implicit in the modeling of the dynamics between the computer system and its agents. In this section the terms computer system, agents, attacks, and intrusions are used in a generic manner. In practice, each IDS must further clarify its threat model in order to be useful to the

[1]Owner is used in a general sense, and refers to the institution interested in detecting intrusions
[2]Note that the connotation of an attacker as sentient is not necessarily valid
[3]Whether or not some agents are capable of having intentions is another issue that is avoided by this thesis

owner of the computer system. Finally, let the term *flaw* refer to an unintended response of the system that is not the direct result of a malicious agent. Examples of flaws include hardware or software errors, environmental effects, and user mistakes.

	Normal Response	Abnormal Response
User (Benevolent Intent)	normal	flaw
Attacker (Malicious Intent)	unsuccessful attack	intrusion

Figure 1-1: Normal and abnormal resposes to an agent

In any real-world system the responses are likely to depend on complex interactions between multiple agents. Furthermore, interactions may produce temporal effects such that the response of a system at any given time depends on events in the distant past. For the moment, assume that a computer system interacts with only one agent at any given time and that its response can immediately be classified as normal or abnormal. Note that this classification of behavior must be made external to the system as it depends on the intentions of the owner.

As a thought experiment, one could construct an optimal IDS in which the computer system alternates between communicating with an unknown agent and its owner. After each potentially malicious interaction, the owner communicates to the IDS whether the system has produced a normal or an abnormal response. Such a setup is obviously impractical for any real-world system. However, it may be reasonable for the owner to classify some subset of the system's responses as normal or abnormal. The role of machine learning in IDS is to take this set of classifications and to produce a classifier that does not require constant interaction with the owner.

1.1.1 Misuse and Anomaly Detection

In a general sense, an intrusion detection system is a means of classifying the system in question as being in one of the four categories shown in figure 1-1. Different means of modeling the system and its input/output characteristics generate the wide range of IDS systems that are used in practice. One of the most general branches in approaches is the distinction between *misuse* and *anomaly detection*. Approaches that attempt to recognize known patterns of activity in system input/output behavior are known as *misuse detection* or template-based systems. Although they are not necessarily implemented in this manner, they can be viewed as consisting of a library of known attacks and a means of deciding if new data belongs to one of the templates. In general such approaches are extremely effective at detecting known attacks but have no ability to detect novel abnormalities. Approaches that attempt to learn normal patterns of behavior and then detect abnormalities in the system response are known as *anomaly detection*. Such systems assume that the responses to novel events differ from some model of normal operations, without specifying the actual deviations *a priori*.

1.1.2 Intrusion Prevention Systems

Rather than attempting to detect intrusions, *intrusion prevention systems* (IPS) take steps to limit the extent to which intrusions can cause damage. Although IPS systems are not directly relevant to the remainder of this thesis, their existence affects the choices about which types of attacks to include in a threat model. In general, IPS and IDS have different strengths and weaknesses. The successful implementation of an IDS suite requires an integration of all available tools for system security. Certain types of attacks are best thwarted through IPS. Examples of IPS include file-system integrity checking, periodic system updating, and forcing information such as passwords to be periodically changed. For a large institutional network, screening incoming emails and file-transfers for known viruses or even for all executables is often performed to reduce the likelihood of the accidental release of a virus. Devices known as *firewalls* block system inputs that violate known protocols or contain suspicious data. A firewall can be thought of as a special form of misuse detection which discards invalid inputs rather than analyzing their effects. The *sandbox* style of designing modern operating systems can also be viewed as a form of IPS in which subsystems have only limited capabilities to interact with other subsystems. Each piece of the operating system is viewed as operating in its own sandbox with a highly arbitrated means of communicating with other sandboxes. Finally, the careful design of software is considered a form of IPS. Designs that use languages such as java, that provide various forms of built-in protection, can theoretically limit the damage that can be caused by an intrusion.

In the real world, large institutions address computer security by employing firewalls, misuse detection, anomaly detection, and a wide range of IPS. Firewalls, IPS and misuse detection are able to prevent many well-known threats and have an excellent record of helping institutions to thwart intruders. However, anomaly detection is the only portion of the defense capable of preventing novel attacks that are unknown to the misuse templates. The remainder of this thesis focuses on anomaly detection.

1.1.3 General Issues for Machine Learning Approaches to Anomaly Detection

Virtually every known machine learning technique has been applied to anomaly detection. However, several important distinctions can be made in their approaches. The types of attackers that are modeled and the particular data that are analyzed make a large difference in the predictive power of the IDS. One important consideration is the granularity at which measurements are made, both with respect to time and to the operations of the computer. Some approaches model time as a continuous variable while others consider only the probability that the system is under attack during a particular time interval such as seconds, minutes or hours. Due to the nature of computer design, the types of measurements that can be made follow a highly hierarchical structure. At the lowest level, one could examine machine instructions executed and the contents of particular memory addresses. At another level, system calls or high-level programming language procedure calls could be recorded. At an even higher level, the aggregate time and memory spent in individual tasks or the amount of hard-drive access or network activity could be analyzed. Various automated methods have been developed for selecting the optimal granularity for a particular domain. In particular S. Raghavan and B. Balajinath discuss a genetic algorithms approach to selecting measurement granularity that could, in principle, be applied to the choices made by hand in this thesis [20].

Traditional machine learning trade-offs such as those between *supervised* and *unsupervised* learning, *off-line* and *online* learning, and *classification* and *regression* all have relevance with respect to anomaly detection. In addition, many types of data pre-processing have been applied to intrusion detection. A recent survey by S. Axelsson gives an excellent overview of existing machine learning approaches to intrusion detection and discusses many of the trade-offs involved [2]. W. Lee and S. Stolfo discuss the systematic application of data-mining techniques to intrusion detection [16]. The trade-offs made in IBS are highlighted and compared to other common techniques where considered appropriate. Pre-processing is discussed at length in the implementation section.

1.1.4 General Operating System Architecture

Processes and the Kernel

In order to motivate the analysis of sequences of system calls, a brief overview of modern computer architecture is given. First, the operations of a computer system consist of the interleaving of instructions generated by separate *processes*. Processes generate a stream of low-level instructions that the computer hardware is able to perform. A typical computer system has two general modes of operation: user mode and kernel mode. A special hardware switch is used to change between modes and control to this switch is highly regulated. A special type of software known as the operating system regulates this access as well as general access to shared resources.

The operating system contains a piece of software known as the *kernel* which is the most trusted component of the system. This piece of software controls access to the kernel-mode switch, which in turn controls access to memory and other hardware devices. A *memory space* refers to the set of memory addresses that a given process is able to legally access. One of the most important roles of the kernel is to ensure that processes do not access memory outside of their memory space, which is regulated by hardware accessible only in kernel mode. In addition to segregating memory spaces, the kernel also allows processes to cooperatively share resources such as the central processing unit (CPU), physical storage devices, network devices, and the display. As part of the memory protection scheme, each process has several regions of memory called *stacks* that control the sequence in which instructions are executed. One of the most common types of intrusions, a *buffer overrun*, involves an attacker causing malicious content to be written onto a process's stack. In this manner, the attacker may be able to cause the process to execute arbitrary instructions when the corrupt portion of the stack is used to determine the location of the next instruction.

System Calls

The operating system provides an *Application Programming Interface* (API) for programs to access shared resources as well as to request that various services be performed. This API consists of a set of *system calls*, which are a means for a normal process to pass control to the operating system in order to perform some restricted activity. Not every process is allowed to call the full range of system calls. A set of programs, known as *privileged*, or *root* processes, is given special permission to make certain system calls. Running a privileged process generally requires that either the operating system itself creates the process for some specialized transient purpose or that a special type of account be used. Such an account is known as the administrator or *root* user. This special account is able to run administrative programs that typical users are generally not allowed to execute. In general, attackers wish to gain the ability to execute such instructions.[4]

Client-server architectures

Two common types of processes are servers and clients. In general, server processes are able to perform activities that a user would not be able to perform directly. For this reason, many server processes operate as a privileged process, and attempt to decide which actions are allowed for particular users. A client process operates on

[4]Or in some cases, to prevent any agent from being able to perform activities, known as a Denial of Service (DOS) attack

behalf of a user and performs the communications necessary to instruct the server process to perform some activity.[5] In general, server processes are passive and perform activities only when instructed by a client process. Furthermore, they generally use a special system call known as `fork` in order to create a child process that services the client.[6] This approach is used in order to allow the parent process to continue to receive new requests as the child process interacts with the client.[7] From a security standpoint, privileged server processes are one of the most vulnerable components as they have privileged access to the machine and also depend heavily on external agents. The computer code related to communications between server and client processes is notoriously difficult to debug and most exploits take advantage of flaws in server processes. For this reason, the remainder of this thesis focusses on privileged server processes.

The goal of many intrusions is to gain root access to a machine in order to perform actions requiring privileged status. While the layer of protection provided by the kernel and the kernel-switch is generally strong, many operating systems provide weaker protection over the privileged status of a process. The means by which an attacker may assume privileged control over the system are generally referred to as an *exploit* as they require taking advantage of an error in the operating system.[8] For this reason, analyzing the behavior of privileged processes is particularly interesting as a means of detecting many types of attacks. The fundamental goal of anomaly detection is to recognize that the behavior of a process has changed. Presumably, when such a change occurs, the owner of the system should be notified or a pre-specified set of actions should be performed. This section has given only a cursory overview of operating system design. More details can be found in standard operating system texts [3, 14, 22, 18].

[5]Note that malicious actions by a client could be caused either by a malicious user, or by malicious code in the client itslf

[6]This is becoming less common with time, as more advanced techniques are developed. However the basic idea of using system calls to support servicing multiple clients remainds the same

[7]In addition, the newly forked process generally sheds as much of its privileged status as possible, in order to reduce the likelihood that it will create a security hole

[8]Taking operating system to include the system programs that run with privileged status

1.1.5 Analyzing Sequences of System Calls

As mentioned in section (1.1.3), the choice of granularity with which system behavior is modeled plays an important role in the effectiveness of an IDS. Given the description above of the role of operating systems and system calls, this thesis takes the stance that sequences of system calls generated by privileged server processes provide an appropriate approximation to system behavior. Unfortunately, for mainly historical reasons, the API provided by most modern Unix systems has become extremely complicated. The behavior of a system call, in general, depends on a number of arguments and even the context in which it is executed. For example, some system calls change the behavior of future system calls by overriding the values of arguments. Some of these complications are addressed during the pre-processing stage discussed in the implementation section. In general, the goal of pre-processing is to create a set of functional states that loosely correspond to the functionally differentiable pieces of the system call API.

On modern Unix systems, system calls are encoded as an integer reference into a large table of functions. These tables have grown to contain about 250 entries. As mentioned above, many system calls are functionally equivalent, and several system calls have very different behavior depending on context or arguments. However the number of functional states is roughly equal to the number of system calls. Therefore, a sequence of integers encoding system calls can be viewed as a rough approximation to the functional behavior requested by the process through the operating system. The degree to which this is a rough approximation is discussed in the implementation section, and again in the experiments section.

In 1996 Forrest, Hofmeyr, and Somayaji wrote a seminal paper on computer immunology [12] suggesting that researchers attempt to mimic natural immune systems in computers. They noted that biological systems are able to distinguish between self and not-self much better than computer systems. Shortly after, they began a series of papers outlining approaches to detecting abnormalities in process behavior based on various observable qualities of the process. They introduced the idea of using sequences of system calls for identification [13], in which they argue that sliding-window based analysis of sequences of system calls could be executed in a massively distributed manner, similar to the operations of a biological immune system. Although the monitor developed in this paper does not use a sliding-window approach, the concept of using sequences of system calls has it roots in these papers.

1.1.6 Comparison of Previous System Call Analyses

In 1999 Warrender et. al. presented a survey of existing methods of analyzing sequences of system calls [23]. They compared the performance of IDS using simple enumeration of sequences, frequency analysis, a rule induction program (RIPPER), and a Hidden Markov Model (HMM). Lee and Helmer each presented alternative rule-based approaches, similar to RIPPER [17, 15]. Unfortunately, they concluded that none of these approaches were clearly an optimal analysis of the system call traces. HMMs have the greatest performance at an extremely high computational expense while less time-consuming methods produce higher false-alarm rates for the same sensitivity. One goal of current IDS research is to develop algorithms with similar performance to HMM but without the extremely high computational costs.

1.1.7 Modeling Server Processes As Markov Chains

In principle, the behavior of a computer process can be viewed as the execution of a *finite state machine* (FSM).[9] As discussed previously, the level of granularity with which states are measured can range over several orders of magnitude. For some appropriate granularity, many processes can be viewed as *Markov chains.* This term is defined rigorously in section (1.3) but loosely means that all information affecting the future behavior of the system is contained within the label of the current state. Server processes generally have a simple input/output centered loop that receives requests and then branches into a different region of the code depending on the type

[9]This is true both in an absolute sense, in which a computer is simply an FSM, as well as in a more practical sense in which states are abstractions used by computer programmers in the design of software

of request. This section avoids probabilities, as they are treated much more rigorously in subsequent sections. However, each of the transitions in a server process can be viewed as occurring with a certain probability that depends on events external to the system. Figure 1-2 shows one means of viewing a simple server process as a Markov chain.

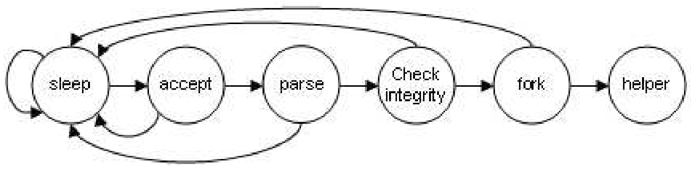

Figure 1-2: A Markov chain representation of a typical server process

The process in figure 1-2 consists of a `sleep-accept-fork` loop. That is, it waits for new connections and then forks off a helper process in order to service each incoming request. If an error occurs in the communication at any step of the process, it reverts back to the sleep state. In addition, steps requiring communications with external agents may involve polling loops, represented in the figure as self-arrows. Several of the steps, most notably the *check integrity* step, may involve privileged access to the operating system. In addition, several of the steps listed above, such as sleep, accept, and fork, correspond to actual system calls. Other states, such as the helper process, may be composed of many system calls. In many cases the helper process is intended to run without privileged status, using the identity of the client.[10] Many exploits involve overriding this protection or taking advantage of flaws in the communication steps earlier in the process.

[10]For example, many sendmail implementations assume the identity of the client in order to write mail to his mailbox and then terminate

1.2 Bayesian Segmentation

As alluded in the previous section, the behavior of a computer process often involves uncertainty generated by the rest of the universe. One branch of mathematics that studies sources of uncertainty is Bayesian statistics. The IBS algorithm is part of a family of *Bayesian segmentation algorithms* that aim to describe a time-series of data in terms of generative probability distributions. In general these algorithms classify various pieces of the time-series as being generated by distinct discrete probability distributions. In other words, they view the sequence as being characterized by a set of distributions as well as a state variable that picks which distribution is active at any given moment.[11] In general, any number of different distributions may be used to generate the series. In addition, the portions generated by each distribution may be interleaved in an arbitrary manner. The term *segmentation* refers to partitioning a time-series. The term *clustering* refers to classifying the segments generated by the segmentation. In other words, segmentation attempts to find break-points in the sequence at which the underlying distribution was changed, while clustering attempts to find segments that were generated by the same underlying distribution. The algorithms' goal, given a time-series, is to produce a set of distributions as well as the break-points where the driving distribution switched between members of the set. The next section provides a brief overview of Bayesian statistics and subsequent sections discuss Bayesian segmentation and clustering algorithms.

1.2.1 Bayesian Inference

Bayesian statistics, in the most general form, provides a framework for combining observed data with *prior* assumptions in order to model stochastic systems. As stated in chapter 2 of *Intelligent Data Analysis* [5], *Bayesian methods are characterized by the assumption that it is meaningful to consider the distribution of parameters defining a distribution, given observed data.* In Bayesian statistics, the parameters defining a

[11]This thesis assumes that all probability distributions are time-invariant. This distinction is not restrictive as an arbitrary number of such distributions is allowed

probability distribution can be estimated. The notation $p(\mathbf{y}|\theta)$ denotes a probability density function (PDF) of the vector of samples $\mathbf{y}$ given a particular estimate θ of the underlying probability distribution generating that data. The parameterization may consist of a single value or a vector of values. For example, a Gaussian distribution is parameterized by two values: $\theta =< \mu, \sigma >$, the mean and variance. Classical statistics allows hypotheses to be generated and tested that relate this PDF to a particular estimate of θ. However, classical approaches do not allow θ itself to be a random variable. Bayesians allow θ to be treated as a random variable by applying Bayes Rule to obtain:

$$p(\theta|\mathbf{y}, I_0) = \frac{p(\mathbf{y}|\theta, I_0)p(\theta|I_0)}{p(\mathbf{y}|I_0)}, \tag{1.1}$$

where $p(\theta|I_0)$ is known as the *prior estimate* of θ and $p(\theta|\mathbf{y}, I_0)$ is known as the *posterior distribution* after considering the evidence $\mathbf{y}$. The prior estimate provides a means of combining exogenous information with observed data in order to estimate parameters of a probability distribution. The denominator, $p(\mathbf{y}|I_0)$ is known as the *marginal density* of the data and does not condition on θ. It can be calculated using the law of total probability as:

$$p(\mathbf{y}|I_0) = \int p(\mathbf{y}|\theta, I_0)p(\theta|I_0)d\theta, \tag{1.2}$$

which integrates over all possible values of θ. (1.2) can be interpreted in a slightly different way, viewing $\mathbf{y}$ as future data and I_0 as available knowledge. If I_k represents all knowledge available until time k and $\mathbf{y}$ is a vector of observations after time k, then (1.2) can be written as:

$$p(\mathbf{y}|I_k) = \int p(\mathbf{y}|\theta, I_k)p(\theta|I_k)d\theta \tag{1.3}$$

For many purposes, it is reasonable to assume that $\mathbf{y}$ depends on I only through the parameter θ. In other words, $\mathbf{y}$ and I are conditionally independent given θ. In this case, $p(\mathbf{y}|\theta, I) = p(\mathbf{y}|\theta)$. This assumption is reasonable when θ can be viewed as completely capturing the state of the stochastic process generating $\mathbf{y}$, such that once the information is used to estimate θ it can no longer provide additional information. If this assumption is made, (1.1) can be re-written as:

$$p(\theta|\mathbf{y}, I) = \frac{p(\mathbf{y}|\theta)p(\theta|I)}{p(\mathbf{y}|I)}, \tag{1.4}$$

where I represents knowledge already incorporated into θ and $\mathbf{y}$ represents new observations. This new θ can then be used to consider future data. The next section discusses an assumption, known as conjugacy, that can often be used to simplify this calculation.

1.2.2 Conjugate Priors

If the prior estimation $p(\theta|I_0)$ has a property known as conjugacy, the quantity $p(\theta|I_k)$ computed from (1.1) is closely related to the prior in that it can be calculated directly without actually applying (1.1). This means that the posterior distribution could be used again as a prior distribution, similar to the original prior in form, but updated to reflect accumulated evidence. As the result of conjugacy, evidence can be added to any posterior distribution in order to form an updated estimate without re-consideration of previous evidence. Although prior conjugacy is simply a mathematical convenience for avoiding recalculations in (1.1), families of distributions with this property can be made sufficiently rich that they can capture a wide range of prior distributions. For this reason priors are often assumed to be of the form of a conjugate distribution. When estimating a single parameter, a Beta distribution is often used. A brief example of repeatedly applying (1.1) to a Beta prior will be described as an introduction to the more complicated *Dirichlet* distribution used in IBS.

The Beta distribution is a family of distributions parameterized by two positive real numbers a and b as defined below:

$$p(\theta|I_0) = B(\theta, a, b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\theta^{a-1}(1-\theta)^{b-1} \tag{1.5}$$

$$Pr\{(I_t = 1)\} = \theta \tag{1.6}$$

$$Pr\{(I_t = 0)\} = 1 - \theta \tag{1.7}$$

where $\Gamma(\cdot)$ represents the well-known gamma function. For example, θ could represent the probability of a particular coin coming up heads when flipped. A Beta distribution has the conjugate property, such that if $p(\theta|I_0)$ has the form $B(\theta|a,b)$ then applying (1.1) will result in a posterior that is also a Beta and is closely related to the prior. Suppose that the information I consists of a series of coin flips and that a heads is represented as a 1 and a tails as a 0. Conjugacy enables the coin flips to be considered incrementally in an attempt to estimate θ, the probability that any future flip is a heads. Note that the assumption that $\mathbf{y}$ is conditionally independent of I given θ is equivalent to the assumption that the probability of flipping a heads is the same for all flips. However, the *estimate* of the probability of flipping heads depends very greatly on the previous flips. The following three equations derive the conjugacy of the Beta distribution. After observing the first flip, with value $I_1 \in \{0,1\}$, applying (1.1) yields:

$$p(\theta|I_1, I_0) = \frac{p(I_1|\theta)p(\theta|I_0)}{p(I_1|I_0)} \tag{1.8}$$

The denominator is obtained from (1.2) as:

$$p(I_1|I_0) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \int p(I_1|\theta)\big(\theta^{a-1}(1-\theta)^{b-1}\big)d\theta, \tag{1.9}$$

where $p(I_1|\theta)$ is given by (1.6) or (1.7). Consider the case in which $I_1 = 1$:

$$p(\theta|I_1 = 1, I_0) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)} \left(\frac{\theta \cdot \theta^{a-1}(1-\theta)^{b-1}}{\left(\frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\right) \int \theta \cdot \theta^{a-1}(1-\theta)^{b-1} d\theta} \right) =$$

$$\frac{\theta^{a}(1-\theta)^{b-1}}{\left(\frac{\Gamma(a+1)\Gamma(b)}{\Gamma((a+1)+b)}\right)} = B\Big(\theta\Big|(a+1), b\Big) \qquad (1.10)$$

By symmetry, if I_1 were a 0, (1.10) would yield $B(\theta|a,(b+1))$. In either case, the posterior distribution is a beta distribution that can be calculated directly from conjugacy, without having to actually apply (1.1). By repeatedly applying (1.10), and writing α_h and α_t in place of a and b, the estimate for the probability of flipping heads after observing h heads and t tails is: $B(\theta|\alpha_h + h, \alpha_t + t)$. Note that the parameters α_h and α_t can be viewed as hypothetical coin flips that occur prior to the experiment.

The section above derived the conjugacy of the Beta distribution when estimating a single parameter. However, in many cases, as is the case in IBS, multiple parameters must be estimated. A conjugate distribution known as the *Dirichlet* is a generalization of the Beta distribution that can be viewed as parameterizing an $n-$sided die rather than a coin. The following notation is used:

$$p(\theta|I_0) = \text{Dir}(\theta|\alpha_1, \alpha_2, ..., \alpha_n) = \frac{\Gamma(\sum \alpha_k)}{\prod \Gamma(\alpha_k)} \prod \theta_k^{\alpha_k - 1}, \qquad (1.11)$$

where $\theta_1, \theta_2, ..., \theta_n$ are parameters that sum to 1. Note that the Beta distribution has $n = 2$ and estimates a single parameter. In general, the condition requiring all probabilities to sum to 1 reduces the number of free parameters to $n-1$. Rather than a single parameter θ representing the probability of flipping a heads, the parameter θ_i represents the probability of rolling a particular face of a die. Similarly, the parameters α_i represent the number of times each face has been rolled. Applying the process outlined in the previous three equations, the posterior probability obtained from (1.1) after observing $r_1, r_2, ...r_n$ rolls of each face is:

$$\text{Dir}(\theta|\alpha_1 + r_1, \alpha_2 + r_2, ..., \alpha_n + r_n)$$

In both the Beta distribution and the Dirichlet distribution, setting each α_i equal to 1 forms a uniform prior. Any Dirichlet distribution with symmetric parameters has a symmetric distribution across the space of all tuples of θ. However, the higher the counts, the less variance in the estimate around a fair die. That is, $\mathrm{Dir}(1, 1, 1, ..., 1)$ has a uniform distribution over all possible dice while $\mathrm{Dir}(100, 100, 100, ..., 100)$ is strongly biased towards a fair die, and symmetric about $1/n$ for each of the parameters.

1.2.3 Modeling a time-series for Bayesian Segmentation

This thesis now shifts focus to developing IBS as a means of analyzing a time-series of data. The length of the series is referred to as N and its members can be modeled as a set of random variables $X_1, X_2, ...X_N$ where $X_i \in \{1, 2, ..., s\}$. The constant s represents the number of discrete states present in the time-series. A particular model M_c describes how the X_i's are related. M_c is a set of discrete probability distributions, along with a mapping from intervals in the time-series to members of the set. This mapping is referred to as a hypothetical sequence C where each element specifies the distribution responsible for generating the corresponding element in S. To give an example, a time-series with 5000 elements may be modeled as being generated by 5 different distributions that each produced intervals of length 1000. Alternatively, it could be modeled by a set of 2 distributions that alternated producing intervals of length 100. As extreme examples, it could be modeled by one distribution that produced 5000 samples or 5000 distributions that each produced 1 sample. The purpose of a *Bayesian segmentation algorithm* is to generate a set of candidate models and to evaluate them according to some criteria, in an effort to provide an estimate of the underlying model.

A model M_c provides a mechanism for determining X_i. The notation x_i is used to represent the actual value drawn from X_i. To avoid confusion between the time-series and the random variables representing the time-series, the letter S is used to represent the actual data as a sequence $S = \{S_i\}$ where $S_i \in \{1, 2, ..., s\}$. Without loss of generality, the sequence can be encoded using the integers between 1 and s. S_i and x_i both refer to the i^{th} element of the time-series. S_i views this number as encoding the state of a physical process while x_i views this number as a sample from the random variable X_i.

1.3 The Markov Property

In full generality each X_i could have a different distribution. Furthermore, X_i could depend on all previous values of x. However, in IBS and related algorithms, models are limited to those that impose the *Markov property* on the X_i's. The Markov property states that the probability distribution X_t depends only on the previous k values $\{x_{t-1}, x_{t-2}, ..., x_{t-k}\}$ for some value of k. The value k is called the *Markov order* of the series. The first major division of Bayesian segmentation algorithms is between those that consider $k = 1$ and those that consider arbitrary values of k. IBS is part of the subset that only allows for $k = 1$, a condition that is typically called Markovian. When $k = 1$ the resulting distribution is called a first-order Markov chain or simply a Markov chain.

The series S is viewed as the concatenation of samples from various Markov chains. An additional parameter θ is necessary in order to specify which of the chains is used to generate a particular X_t. The parameter θ represents the hypothetical members of C as random variables. For IBS, the Markov condition on the X_i's is equivalent to:

$$\forall t > 0, \quad Pr\{(X_t = j|(x_0, x_1, ..., x_{t-1}), \theta)\} = Pr\{(X_t = j|x_{t-1}, \theta_{t-1})\}. \tag{1.12}$$

A Markov probability distribution can be represented concisely as a matrix of state transition probabilities. The entry $M[i][j]$ contains $Pr\{(X_t = j|x_{t-1} = i)\}$. M_c can be represented as an array of matrices indexed by θ such that:

$$M_c[t] \equiv Pr\{(X_t = j|x_{t-1}, \theta_{t-1})\} = M_{\theta_{t-1}}[x_{t-1}][j], \tag{1.13}$$

where M_θ indicates the θ^{th} matrix in the array M. This equation defines a PDF over possible values of X_t that depends on the previous state and the cluster membership at time $t-1$. Note that in M_θ the rows sum to one and have non-negative components. In general, the notation M_c refers to both the array of Markov matrices and the vector θ. In other words, M_c completely parameterizes the X_i's. Therefore, the role of a Bayesian segmentation algorithm is to produce an M_c given a time-series S. Note that the distinction between segmentation and clustering can be viewed as a transformation of one M_c to another in which several (possibly not-contiguous) distinct segments are combined to have the same value of θ.[12]

1.4 Bayesian Model Selection

Given the constraint on M_c that the X_i's be Markovian, the universe can be viewed as consisting of some set M of Markov processes and an oracle[13] that chooses when and for how long each process is allowed to generate samples. Bayesian segmentation algorithms attempt to find some set of matrices $\hat{M}$ that approximates the true set M, as well as to estimate the break-points produced by the oracle. Note that estimated break-points correspond to points at which $\hat{\theta}$ changes values. The selection of M_c was first viewed as a Bayesian model selection problem by the creators of IBS [6] and the remainder of this section outlines the derivations presented in that paper.

1.4.1 Off-line and On-line Model Selection

A Bayesian approach to model selection compares two possible models M_1 and M_2 in terms of their *likelihood* given the data. Let $p(M_c|S)$ be the likelihood of the particular model M_c given the sequence of observations S. This term is at the heart of Bayesian segmentation and appears in most of the equations derived below. In general these algorithms can be grouped into *off-line* and *on-line* versions. Off-line variants are allowed to consider the entire sequence S of data while on-line variants attempt to approximate the solution by incrementally considering new data. Both

[12]In many contexts this transformation is known as *lossy-compression*, but the relationship between segmentation, clustering, and compression is not discussed any further in this thesis

[13]An oracle is a source of randomness that is external to the system and observeable only through S

types of analysis are important to fully describe IBS.

IBS is an on-line approximation to *Bayesian clustering by dynamics* (BCD), which is a *Bayesian clustering Algorithm.* Clustering algorithms generally start with a set of segments and attempt to group similar segments together, rather than attempting to produce the clusters directly from the original sequence. In other words, by convention, clustering algorithms assume that the sequence has already been segmented and we evaluate their performance independent of the performance of the segmentation. Before discussing IBS in detail, an optimal clustering algorithm is derived. Next, BCD is presented as an approximation to the optimal algorithm. Finally, IBS is derived as an approximation to BCD.

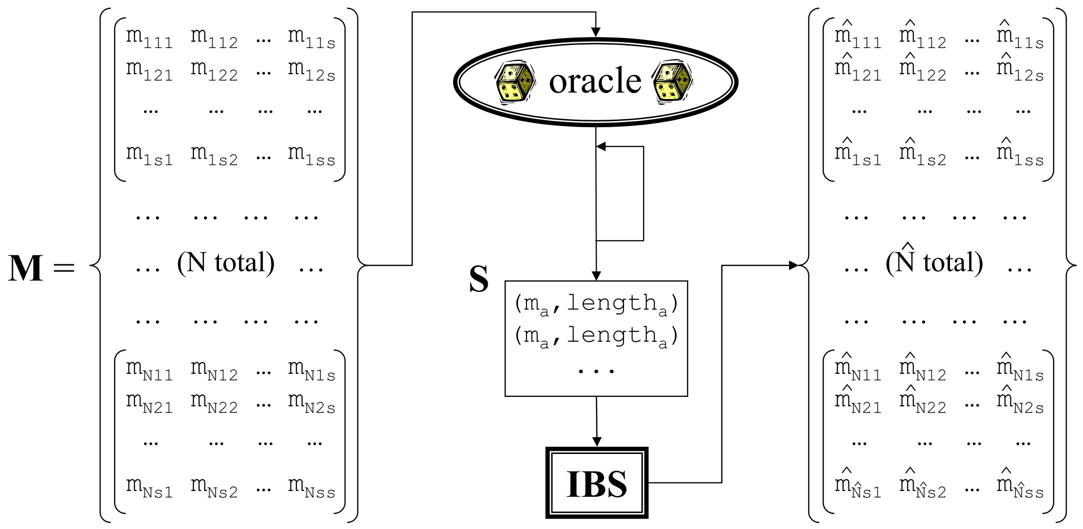

Figure 1-3: An oracle produces S using the set M of Markov matrices. Bayesian segmentation algorithms attempt to reverse-engineer M.

1.4.2 Partitioning and Clustering

The information encoded in θ can be viewed as a type of partioning of S into equivalence classes. θ maps each element of S to a member of the set of Markov matrices.[14] Note that there must be between 1 and N different matrices. θ is by definition piece-

[14]It is really a mathematical partition combined with a mapping from partitions to one of the matrices within the model

wise constant. Also note that the distinction between segmentation and clustering can be viewed as whether θ can be written as a monotonic sequence or if it requires returning to previously used values.

Subsequences of S that are assigned to the same equivalence class are known as *executions* or *episodes*. This terminology emphasizes the fact that an oracle chooses a stochastic process and allows it to generate samples. Let S_θ refer to the set of subsequence of S such that every member is mapped to the same matrix within M_c. This notation is defined more formally in the next section. Informally, S_θ contains all of the executions assigned the θ^{th} matrix in $\hat{M}$.

1.4.3 Executions and Clusters

Unfortunately, the definition of an execution is slightly complicated by boundary conditions. Note that in (1.13) X_t depends on θ_{t-1}. Therefore the transition between the last element of one execution and the first member of the next execution is assigned to the first execution's matrix. That is, at the point at which the oracle chooses a new matrix, the value of θ changes. However, the transition between the previous state and the new state depends on the old value of θ. Similarly, an execution does not contain a transition into its starting state. The definition of S_θ is therefore amended to include the boundary-case conditions. In order to specify that the transition into a state is included, but not the transition out of that state, S_θ contains an extra 0 after the first element of the next execution. Also, let S_θ contain the concatenation of all executions that share the same value of θ within S. The concatenation preserves the artificial 0's. In other words, S_θ is a sequence containing the transitions hypothesized to have been generated while the oracle was using the matrix $\hat{M}_\theta$.

The definition of S_θ is illustrated with an example. In figure 1-4, $A, B, ..., H$ are executions, and (AB) is also an execution, demonstrating that any subsequence of an execution is also an execution. In this model there are 3 equivalence classes. Viewing the model $M_c = \{M_1, M_2, M_3\}$ as generating S, $\{A, B, D, G\}$ is a cluster, $\{C, E\}$ is another cluster, and $\{F, H\}$ is the final cluster. S_{θ_1} would be the sequence containing A then the first element of B, followed by a 0, all of B, followed by the first element

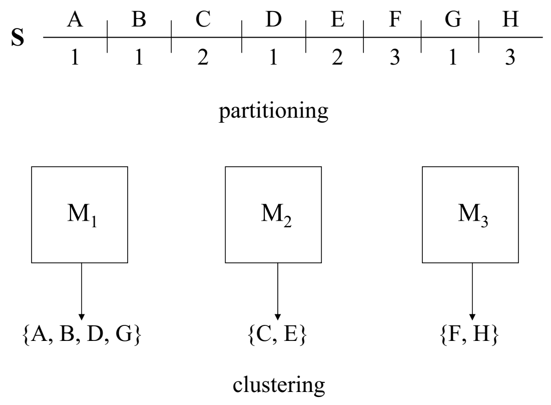

Figure 1-4: Segmentation of S into 8 segments and clustering into 3 clusters

of C and then a 0. The next element would be the first element of D. After the first element of E a 0 would follow and the next element would be the start of G. S_{θ_1} would conclude with the end of G. Ignoring transitions into or out of 0, the transitions contained by S_{θ_1} are the same as those generated while M_1 was used by the oracle. Note that the fact that A and B were actually contiguous was preserved by the fact that both the transition into and out of the first element of B were included once. The process of partitioning S into $A, B, , ..., H$ is known as segmentation and the process of grouping $\{A, B, D, G\}$ is known as clustering.

1.5 An Optimal Clustering Algorithm

In principle, one could generate all possible partitions of S. There are $g(n)$ of them, as defined below, and each one implies a particular θ. One could enumerate all possible partitions, generate the best model given each partition, and then select the globally optimal model through an exhaustive search. However, the total number of ways to partition S is related to the Stirling numbers of the second kind and is given by:

$$g(n) = \sum_{k=1}^{n} \frac{1}{k!} \sum_{i=0}^{k-1} (-1)^i \binom{k}{i} (k-1)^n \tag{1.14}$$

This quantity (1.14) is known to be super-polynomial. Clustering algorithms are heuristic searches through this space of all possible partitions.[15] However, before discussing BCD, it is necessary to derive the means of producing a model given a partition. Furthermore, a method for comparing the models generated by two different partitions is required. The next several sections provide a framework for answering these questions.

1.5.1 Maximum Likelihood Matrices

The first step in solving the optimal clustering problem is deriving a process for producing a model, given a particular partition. Encoding the executions $S_{\theta_1}, S_{\theta_2}, ..., S_{\theta_c}$ as Markov matrices requires the estimation of c matrices. The Markov assumption implies that each matrix only depends on the transitions to which it is assigned. As likelihood will later be used to compare models, the maximum likelihood estimate for each matrix can be shown to generate the best possible model, given a partition. The transitions stored in S_θ can be stored in a *count matrix* which is an $s \times s$ matrix N_θ in which n_{ij} is the number of times that a transition from state i to state j occurred in the execution.[16] One implication of the Markov assumption is that the storage required for a cluster depends only on s^2 and not on the number of transitions.[17]

The optimal transition probability matrix for generating a particular count matrix is formed by normalizing each of the rows as follows:

$$\hat{P} = (\hat{p}_{ij}) = \frac{n_{ij}}{\sum_j n_{ij}}. \tag{1.15}$$

[15]Assuming that each partition can be mapped to an optimal clustering, which is discussed in the next section

[16]Parsing the artificial 0's as described above

[17]Assuming that none of the entries would overflow, which is addressed briefly later in the paper

For example, referring back to figure 1-4, if $s = 5$, $C = \{1, 2, 3, 4, 5, 1\}$ and $E = \{5, 1, 3\}$, then the count matrix and associated probabilities are:

$$N_{AC} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 2 & 1 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 & 1 \\ 5 & 1 & 0 & 0 & 0 & 0 \end{array} \Rightarrow \hat{P}_{AC} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 2/3 & 1/3 & 0 & 0 \\ 2 & 0 & 0 & 1/1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 1/1 & 0 \\ 4 & 0 & 0 & 0 & 0 & 1/1 \\ 5 & 1/1 & 0 & 0 & 0 & 0 \end{array}$$

Figure 1-5: A count matrix and the induced transition probabilities

1.5.2 Bayesian Parameter Estimation

(1.15) can be extended, as motivated in the introduction to Bayesian analysis, to include prior knowledge of the probabilities for each of the transitions. This Bayesian approach not only allows for the incorporation of exogenous data, but also prevent 0's from occurring in the transition probability matrices (as they do in figure 1-5).[18] Prior knowledge is incorporated by creating a hypothetical time-series of length $\alpha + 1$ in which the α transitions create a hypothetical count matrix with entries α_{ij}. The count matrix N is formed by adding α_{ij} to each of the entries. The new estimate for P is known as the *Bayesian estimate* and has:

$$\hat{P} = (\hat{p}_{ij}) = \frac{\alpha_{ij} + n_{ij}}{\sum_j (\alpha_{ij} + n_{ij})}. \tag{1.16}$$

Suppose that the example given above had a uniform prior with every $\alpha_{ij} = 1$.[19] The modified count table would be:

The α's are known as hyper-parameters and are viewed as free parameters for the algorithm. The are generally domain-dependent but viewed as constants within a particular domain.

[18] A probability matrix with 0's is troublesome because it is not known if the oracle will eventually provide a sequence that would be classified using a 0-probability transition

[19] This prior is the $\text{Dir}(\theta|1, 1, 1..., 1)$

$$N_{AC} + N_{\alpha} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 1 & 3 & 2 & 1 & 1 \\ 2 & 1 & 1 & 2 & 1 & 1 \\ 3 & 1 & 1 & 1 & 2 & 1 \\ 4 & 1 & 1 & 1 & 1 & 2 \\ 5 & 2 & 1 & 1 & 1 & 1 \end{array} \quad \Rightarrow \hat{P}_{AC} = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 1/8 & 3/8 & 2/8 & 1/8 & 1/8 \\ 2 & 1/6 & 1/6 & 2/6 & 1/6 & 1/6 \\ 3 & 1/6 & 1/6 & 1/6 & 2/6 & 1/6 \\ 4 & 1/6 & 1/6 & 1/6 & 1/6 & 2/6 \\ 5 & 2/6 & 1/6 & 1/6 & 1/6 & 1/6 \end{array}$$

Figure 1-6: The count table and induced probabilities with a prior estimate

1.5.3 A Bayesian Approach to Model Evaluation

Using the formulas derived above, an optimal model can be generated from any partition of S. All that remains to be derived for an optimal algorithm is a means of comparing two models. A Bayesian method for comparing models is now developed. The first step applies (1.1) to the likelihood of a model to obtain:

$$p(M_c|S, I_0, \alpha) = \frac{p(S|M_c, \alpha)p(M_c|I_0)}{p(S|I_0)} \tag{1.17}$$

The denominator $p(S|I_0)$ is constant for each of the prospective models. In order to maximize (1.17) the numerator must be maximized. $p(M_c|I_0)$ is a measure of the *a priori* probability for the particular clustering M_c. Some Bayesian clustering algorithms treat this calculation as a free-parameter in order to favor different types of clustering. However, without any reason to favor a particular clustering, this quantity can also be considered a constant. Section (1.9) discusses the consequences of this assumption. For now, this quantity is treated as a constant and applying (1.13) gives an optimal model that maximizes:

$$p(S|M_c, \alpha) = \prod_{\theta} \prod_{t \in t_\theta} \hat{P}_\theta[S_t][S_{t+1}]. \tag{1.18}$$

While mathematically correct, the above formulation is subject to underflow as the result of multiplying many small numbers. Logarithms can be applied to the terms in the product above without changing the maximization process. As the result, the optimal clustering algorithm enumerates over all possible partitions, generating an optimal model for that partition according to (1.16), and chooses the model that maximizes:

$$l(S|M_c,\alpha) = \log(p(S|M_c,\alpha)) = \sum_{\theta}\sum_{t\in t_\theta}\log(\hat{P}_\theta[S_t][S_{t+1}]). \tag{1.19}$$

The optimal algorithm described above is clearly super-polynomial. BCD is an attempt to apply a heuristic search to the space of all possible models in order to make the algorithm more feasible.

1.6 Bayesian Clustering by Dynamics

BCD operates in two phases. In the first phase a break-point detection algorithm is applied to S. The algorithm itself is not specified by BCD, but a particular variant used in IBS is discussed in (1.7.3). The break-point detection algorithm produces a set of segments that are stored as count matrices with the possible addition of priors.

The second phase of BCD takes a set of segments and attempts to form an optimal set of clusters by combining segments. In other words, it modifies the θ vector by taking segments with different values of θ and assigning them the same value. This process is known as subsumption and has the property that the count matrix of the subsumed cluster is the sum of the count matrices for each of the original clusters. In practice θ is not stored explicitly as only count matrices are ever needed for the actual computations.

The process of combining segments to form clusters is agglomerative, meaning that segments are never broken apart but can be subsumed together. Finding the optimal agglomeration of segments into clusters forms a much smaller solution space than the set of all models considered in the optimal algorithm, but is again intractable. BCD uses a heuristic search through the space of possible subsumptions to find an approximately optimal solution.

The heuristic search has three important components: a metric for evaluating whether a particular subsumption improves the model, a method of generating potential pairs to subsume, and a termination condition. Various choices for these criteria give rise to a family of related algorithms.

1.6.1 Evaluating a clustering

As in the optimal clustering algorithm, the parameter optimized in BCD is $l(S|M_c)$. However, rather than evaluating all possible models, this score is used to incrementally improve the clustering. Given the current set of clusters $M_c = \{M_1, M_2, ..., M_c\}$ the algorithm tests whether some particular pair of clusters M_i and M_j should be combined into M_{ij}. The clusters M_i and M_j are subsumed if:

$$l(S|\underbrace{\{M_1, ..., M_{ij}, ..., M_c\}}_{c-1}) \overset{?}{>} l(S|\underbrace{\{M_1, ..., M_i, ..., M_j, ..., M_c\}}_{c}) \tag{1.20}$$

In the optimal algorithm, $l(S|M_c)$ was calculated directly from S and θ. However, the authors of BCD show that it can be calculated from count matrices as well[6]. The derivation given in the original paper is outlined in the remainder of this section.

Expression	Meaning
$S \equiv \{S_1, ..., S_k, ..., S_c\}$	executions having same transitions
$S_k \mapsto N_k \mapsto \hat{P}_k$	cluster produces counts and a Markov matrix
$\alpha_{kij} = \alpha_{ij}$	in cluster k
$n_{kij} = n_{ij}$	in cluster k
$\alpha_{ki} = \sum_j \alpha_{kij}$	row precision in cluster k
$n_{ki} = \sum_j n_{kij}$	occurrences of i in cluster k
$m_k = \sum_i n_{ki} = \|S_k\|$	length of cluster k
$m = \sum_k m_k = \|S\|$	length of S
$\alpha = \sum_k \alpha_{ki}$	global precision
$C = \{C_1, ..., C_m\}$, $C_i \in \{1, 2, ..., c\}$	$C_i = j$ if S_i is a member of cluster j

Figure 1-7: Definitions used in the derivation of BCD

The quantity α is called *global precision* and represents the influence of the prior estimates on the probabilities. The remainder of this paper assumes that the α_{kij} follow a Dirichlet distribution. If the priors did not have conjugacy, each transition would require re-calculating equations based upon (1.1) and (1.2). Referring back to (1.12), $p(X_t = j|x_{t-1})$ is conditionally independent of t given C_{t-1}. In other words,

C_{t-1} can be viewed as a state variable, specifying which model to use to generate X_t given x_{t-1}.

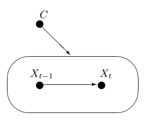

Figure 1-8: X_{t-1} and X_t are conditionally independent given C_{t-1}, taken from [6]

This observation, first made in 1992 by Cooper and Herkovits [7] motivates writing $p(S|M_c)$ as a function of S and C as follows:

$$p(S|M_c) = f(S, C)g(S, X_{t-1}, X_t, C). \tag{1.21}$$

Where:

$$f(S, C) = \frac{\Gamma(\alpha)}{\Gamma(\alpha + m)} \prod_{k=1}^{c} \frac{\Gamma(\alpha_k + m_k)}{\Gamma(\alpha_k)}. \tag{1.22}$$

This expression comes from the assumption that the lengths of clusters obey a Dirichlet distribution. Note that this expression does not depend on actual values within S but only on the number and relative lengths of the clusters. Treating the α's as a Dirichlet probability distribution, this term gives the *a priori* likelihood that the sequence would be partitioned in k clusters each with length m_k. Alternatively, this can be viewed as a penalty term that discourages S from being overly partitioned. In this light, this term is an application of Occam's razor, causing BCD to favor the smallest number of free parameters that adequately describe the sequence.

$$g(S, X_{t-1}, X_t, C) = \prod_{k=1}^{c} \prod_{i=1}^{s} \frac{\Gamma(\alpha_{ki})}{\Gamma(\alpha_{ki} + n_{ki})} \prod_{j=1}^{s} \frac{\Gamma(\alpha_{kij} + n_{kij})}{\Gamma(\alpha_{kij})}. \tag{1.23}$$

Here the first product is over the set of clusters, the second is over the rows of the associated count matrices, and the third is over each of the entries in that row. This formula comes from the assumption that the parameters in the c Markov matrices follow Dirichlet distributions which follows from applying Bayes rule to a conjugate distribution.

Unlike the optimal algorithm, BCD evaluates likelihood solely on the basis of the priors and the generated count matrices. The equations above are converted into their log form for actual numerical computation. This conversion introduces the log-gamma function, lgf, leading to:

$$\mathrm{lf}(S, C) = \frac{\mathrm{lgf}(\alpha)}{\mathrm{lgf}(\alpha + m)} \sum_{k=1}^{c} \frac{\mathrm{lgf}(\alpha_k + m_k)}{\mathrm{lgf}(\alpha_k)}. \tag{1.24}$$

$$\mathrm{lg}(S, X_{t-1}, X_t, C) = \sum_{k=1}^{c} \sum_{i=1}^{s} \frac{\mathrm{lgf}(\alpha_{ki})}{\mathrm{lgf}(\alpha_{ki} + n_{ki})} \sum_{j=1}^{s} \frac{\mathrm{lgf}(\alpha_{kij} + n_{kij})}{\mathrm{lgf}(\alpha_{kij})}. \tag{1.25}$$

$$l(S|M_c) = \mathrm{lf}(S, C) + \mathrm{lg}(S, X_{t-1}, X_t, C). \tag{1.26}$$

1.6.2 Generating a Space of Potential Models

In BCD the calculation of $l(S|M_c)$ is performed on only a subset of all possible models. Given an initial segmentation, it calculates a pair-wise distance between each pair of distributions and builds a sorted list of all such pairs. The first two entries have the globally smallest distance, and every subsequent pair has the next smallest distance. Note that each matrix appears $c - 1$ times in this list. The distance used by BCD is based on the Kullback-Leibler distance (relative-entropy):

$$\mathrm{d}(p_1, p_2) = \sum_{i=1}^{s} p_{1_i} \log \frac{p_{1_i}}{p_{2_i}} \tag{1.27}$$

This equation is not necessarily symmetric, but can me made into a valid metric by calculating:

$$\mathrm{KL}(p_1, p_2) = \frac{\mathrm{d}(p_1, p_2) + \mathrm{d}(p_2, p_1)}{2} \tag{1.28}$$

BCD walks along the sorted list and tests whether subsuming the first two matrices would produce a better score according to (1.26). If the matrices are subsumed, it then re-scans the list removing the original matrices and merging in pairs including the newly created matrix. If the resulting score of the subsumption is lower, it does not perform the subsumption, drops the pair from the list, and moves to the next pair. BCD terminates when it does not find any pair that can be subsumed to increase the score.

1.6.3 Complexity and Effectiveness of BCD

The implementation of the BCD algorithm involves several important decisions. First, the method of choosing an initial partition is extremely important (and not specified by BCD itself, as it is a clustering algorithm and not a segmentation algorithm). Additionally, if a particular number of clusters is desired, this initial partition can be chosen to optimally partition the space. In many cases, if clusters are short, they will be subsumed with nearly any other short cluster, meaning that the order that this space is searched can have a large effect on the final clustering. Finally, the choice of priors and their relative weights can bias the algorithm towards particular ends of the solution space. The authors point out that uniform priors encourage clusters to be subsumed into a single cluster despite the intuitive notion that they are uninformative. They also prove that the time complexity of the algorithm is $O(c^4s^2)$ where s is the number of total unique states and c is the number of initial clusters. The IBS algorithm is an attempt to approximate the behavior of BCD while operating in an on-line fashion, meaning that the average incremental work required when given a new sample from S is $O(1)$.

1.7 The IBS Algorithm

1.7.1 IBS Motivation

Unlike the optimal clustering algorithm and BCD, the IBS algorithm attempts, in theory, to operate on an infinite sequence of data. Rather than viewing S as a data-set, it views the sequence as samples from some infinite process. In addition, the samples are generated at a particular time frequency such that IBS must be able to completely process a state transition before receiving the next data point. In principle a buffer could be employed to smooth any temporal variations in processing speed, but this buffer must be finite. While BCD was given the task of explaining S by some agglomerative combination of clusters, IBS is responsible for both segmenting the series and then grouping the segments into related clusters, all in amortized constant time.

The motivation behind IBS is rooted firmly in a Bayesian view of the world. If the oracle producing the infinite sequence has a finite set of time-invariant Markov processes, and he uses a time-invariant probability distribution to choose the order in which the processes execute, and the execution time itself is produced according to a time-invariant probability distribution, then maximum likelihood estimates for each of those parameters have a logical interpretation. Unfortunately, it is not feasible to allow all of those parameters to range over arbitrary values. Instead IBS relies on prior estimates for each of the parameters in the traditional Bayesian fashion. In particular, the matrices are assumed to follow Dirichlet distributions,[20] the selection by the oracle is assumed to be uniform across the set M,[21] and the probability of a break-point is assumed to be constant, yielding an exponential distribution over time between break-points. Also, the data available for IBS at any given transition is only the most recent sample from the time-series, the current count matrix, and the library of existing clusters. All other knowledge must be contained in the updated posterior estimates. In other words, with each transition, IBS incorporates the knowledge into its prior assumptions about the next transition and can never undo the effects of this calculation.[22]

[20]The Dirichlet family is sufficiently rich that this assumption asymptotically approaches the true set M with infinite data

[21]If a different assumption were made, $p(M_c|I_0)$ could be included in the optimization, instead of treated as a constant

[22]One potential strategy to combat early mistakes would be to prune away clusters that have very few transitions after a suitable length of time, but this technique is not implemented or further mentioned

1.7.2 Structure of IBS

IBS consists of several tiers of nested loops. At the highest level, it processes samples from the infinite sequence and runs a break-point detection algorithm. Each time it decides that it has reached a break-point, it attempts to classify the latest segment. This classification must consider subsuming the latest segment into each of the previous clusters. In addition, it must test the likelihood that the segment has come from a previously unobserved process. In theory, if the number of distinct processes were known, or conjectured, the probability that a new segment is novel could be made conditional over all of the finite values for the number of processes. However, in general, it is difficult to parameterize this distribution and $p(M_c|I_0)$ is assumed to be a constant.

As a further complication, it is not clear whether the probability of seeing a new cluster is constant or decreases with time. If the oracle is indeed using a time-invariant method of choosing which process to execute then intuitively this should be a decreasing function and this fact should be accounted for. If instead the oracle is viewed as constantly generating new matrices, or using a time-dependent means of selecting which matrix to execute, this parameter should be constant. The remainder of this thesis assumes that this parameter is constant, rather than decreasing, which allows the likelihood calculations derived in BCD to be used without modification.

In general, without making assumptions about the distinct probability distributions used by the oracle, it is difficult to know whether the distance between two estimated probability distributions is due to a fundamental difference in the driving process or in sampling variation.[23] The current version of IBS assumes that the set M of matrices is generated to uniformly sample the space of all $s-$dimensional matrices. By assuming that the true set M uniformly samples the space of all possible Markov matrices, the algorithm looses the ability to discern between matrices that are close in probability space. However, this difficulty can be overcome by pre-loading the library

[23]For example, consider observing 100 coin flips with 51 heads and 49 tails. The MLE for the probability of heads would be 0.51. However, conditional on the probability being .50, the probability of observing this data would still be very high

with known matrices if the assumption is invalid for a particular domain.[24]

1.7.3 Break Point Detection

The outer loop of IBS runs a break-point detection algorithm designed to divide the time-series into segments. The break-point algorithm works by accumulating the cumulative probability of the sequence and breaking when it exceeds certain bounds. The bounds are based on the expected value of the cumulative probability, conditional on the assumption that the probabilities being used are correct. In other words, a particular model is used both to accumulate probabilities and to accumulate expectation. As long as the assumption that the model is accurate holds, these quantities should, on average, stay within a reasonable distance. However, if the assumption is wrong and a new process is driving the samples, the distance should diverge quickly (assuming that the two matrices are not too close in probability space).

At each transition three variables are updated: the actual cumulative probability, the estimated expected value of this probability, and the estimated variance of this quantity. A break-point occurs when the actual cumulative probability deviates from the expected value by more than β standard deviations. A closed form for the optimal value of β is not known and section (3.3) attempts to use empirical evidence to determine an appropriate value for various sizes of matrix.

A major consideration in the break-point detection is how to estimate the probability of a particular transition. This choice is made by building a count matrix for the current segment, including a hypothetical sequence of priors, and using (1.16). For each transition, the corresponding entry in the $\hat{P}$ matrix is used as an estimate of the probability of that transition occurring. A variable known as `score` is updated by adding the log of $\hat{P}[x_{i-1}][x_i]$. `score` is known as the log-likelihood of the sequence, although the fact that $\hat{P}$ changes at every point makes this value an estimate. Simultaneously, the expected value and variance of this parameter are updated using the

[24]For example, if it were known that two types of dice existed: fair dice and dice with a .60 probability of flipping a heads, pre-loading the library with these two matrices would help a sequence with 58 heads and 42 tails be classified in the .60 category rather than being subsumed with another sequence containing a 50-50 split

standard definition of expectation and variance.

$$c = -\log(\hat{P}_{t-1}[x_{t-1}][x_t]) \tag{1.29}$$

$$E[c] = \sum_j p(c_j)c_j = -1 \cdot \sum_j (\hat{P}_{t-1}[x_{t-1}][j]) \cdot \log(\hat{P}_{t-1}[x_{t-1}][j]) \tag{1.30}$$

$$\begin{aligned} \mathrm{var}(c) = \sum_j p(c_j)(c_j - E[c])^2 = \\ \sum_j (\hat{P}_{t-1}[x_{t-1}][j]) \cdot (-\log(\hat{P}_{t-1}[x_{t-1}][j]) - E[c])^2 \end{aligned} \tag{1.31}$$

$$\mathrm{score}_t = -1 \cdot \sum_{i=1}^{t} \log(\hat{P}_{i-1}[x_{t-1}][x_t]) \tag{1.32}$$

$$\mathrm{mean}_t = -1 \cdot \sum_{i=1}^{t} \sum_{k=1}^{s} \hat{P}_{i-1}[x_{t-1}][k] \cdot \log(\hat{P}_{i-1}[x_{t-1}][k]) \tag{1.33}$$

$$\mathrm{variance}_t = \sum_{i=1}^{t} \sum_{k=1}^{s} (-\log(\hat{P}_{i-1}[x_{t-1}][k]) - \mathrm{mean}_t)^2 \cdot (\hat{P}_{i-1}[x_{t-1}][k]) \tag{1.34}$$

$$\mathrm{sd}_t = \sqrt{\mathrm{variance}_t} \tag{1.35}$$

The break-point criteria is to break if:

$$|\mathrm{score}_t - \mathrm{mean}_t| > \beta \mathrm{sd}_t \tag{1.36}$$

Each time a break-point is triggered, the count matrix used to generate $\hat{P}$ is then passed to the clustering stage of the algorithm.

1.7.4 IBS Clustering

The clustering phase of IBS is similar to the clustering performed in BCD. However, rather than taking a set of segments as an argument, IBS considers each segment incrementally. It generates a likelihood score in which the latest segment is added as a novel cluster against the set of scores obtained by subsuming the cluster into each existing cluster. Previous discussions about the probability of observing a novel cluster and the method used by the oracle to choose which matrix to execute are both important for the calculation of these scores. In the actual implementation, the probability of observing a novel cluster is viewed as constant with respect to time and the probability that the oracle chooses a particular matrix is uniform over the set of matrices. However, more complicated calculations using Bayesian mixing could be used at the clustering phase to incorporate a different set of assumptions.

1.7.5 An Alternative Clustering Scheme

As a potential alternative to the clustering scheme described in BCD, a standard non-Bayesian clustering algorithm was also implemented. Rather than clustering based upon likelihood as defined in (1.17), clustering is performed based upon a *minimum description length* (MDL) for the data. In such a scheme, the subsumption of two clusters is performed if the combined cluster can be described using fewer bits of information than the sum of the two clusters. Let S_1 and S_2 be two sequences of lengths N_1 and N_2 and $\hat{\theta}_1$ and $\hat{\theta}_2$ be the estimated probability distributions used to describe them. Let d represent the *degrees of freedom* of $\hat{\theta}$. The MDL for S_1 and S_2 is defined as:

$$DL(S_1)+DL(S_2) = -\log(P(S_1|\hat{\theta}_1))+\frac{d}{2}\log(N_1)+-\log(P(S_2|\hat{\theta}_2))+\frac{d}{2}\log(N_2) \quad (1.37)$$

If S_1 and S_2 are subsumed, creating the sequence S_{12} and a new probability distribution $\hat{\theta}_{12}$, its new MDL is defined as:

$$DL(S_{12}) = -\log(P(S_{12}|\hat{\theta}_{12})) + \frac{d}{2}\log(N_1 + N_2) \quad (1.38)$$

The alternative clustering scheme performs subsumptions that produce a smaller MDL. The intuition behind this clustering is that $\hat{\theta}$ could be used, in an information-theretic sense, to encode the sequence S using $-\log(P(S|\theta))$ bits. However, as $\hat{\theta}$ is a maximum likelihood estimate, the value of $P(S_{12}|\theta_{12})$ will always be greater than $P(S_1|\theta_1) + P(S_2|\theta_2)$. In order to correct for this effect a penalty term is added to each of the description lengths. The term $\frac{d}{2}\log(n)$ is known as the Bayesian information criteria (BIC) and is one possible choice of penalty terms. Several logical choices exist for d. If the matrices are assumed to be a uniform sampling as discussed in (1.9), then a value of $s(s-1)$ would be appropriate as the matrices have full flexibility. If, however, attacks are assumed to differ from normal activity only with respect to a single system call, then a value of s would be more appropriate. Finally, if attacks are assumed to differ from normal behavior only in a single type of transition, then a value of 1 would be appropriate.

1.7.6 IBS Summary

For the practical application of IBS to a real-world problem, a number of issues need to be considered. In particular, the value β of the break-point detection threshold and the usage of priors are free-parameters. In addition, preloading the library of matrices with domain-specific distributions can be necessary if the sequences are too short to fully characterize the underlying probability distribution. Also, the length of sequences needed grows as a function of s because the number of independent parameters in each matrix grows. In general, the longer the length of each execution, the more power IBS has to differentiate between clusters. Also, if executions are short and the information content is dominated by priors, all resulting matrices look similar and are subsumed together.

1.8 Applying IBS to IDS

In order to apply IBS to sequences of system calls a number of decisions must first be made. First, the encoding of a trace of system calls into a suitable sequence S may include several forms of pre-processing. Next, the methods of training the algorithm to optimally detect intrusions must be investigated. Finally, the algorithm must be interfaced with suitable controls in order to make it a useful part of an IDS suite. The remainder of the introduction focuses on these practical issues and presents the development of a hypothetical IDS suite using IBS for anomaly detection.

Ideally, an IDS built for the real-time monitoring of system calls would be integrated into the system kernel in order to have access to the largest set of knowledge about system state and the exact order of system call execution. However, as a close approximation, most operating systems provide a utility by which the system calls generated by a particular process and its child processes may be logged to a file. For example, the SunOS operating system contains a utility known as truss that was used to generate the data-sets examined in the experiments section. Most versions of linux have a similar utility known as trace or strace. In general, running such a process requires root access. The actual implementation of IBS requires several preprocessing steps discussed in the implementation section. Next, matrices may be chosen to preload the library used for clustering. If known patterns of attacks are available, either by off-line analysis or by the results of previous trials, they can be added to the library along with a flag that denotes that they are abnormal patterns. In addition, normal behaviors corresponding to different pieces of code, or different types of operation could be used.[25] Preloading of the library helps IBS to reach a more accurate description of the world as initial data points are not accidentally clustered together.

A practical IDS would require that the output from IBS be filtered for interesting events. Unfortunately, the definition of interesting must be made external to IBS due to the great diversity of types of computer systems and the degree to which one is interested in their security. One strong benefit of IBS is the fact that abnormal activities that are detected and reported can easily be laballed as normal clusters such that subsequent activity that was attributed to that cluster would no longer trigger an alert.

1.9 Random Probability Distributions

As mentioned in the derivation of $p(M_c|S, I_0)$, the implementation of IBS makes the assumption that the Markov matrices generated by the oracle are uniformly distributed over the space of all such matrices. In other words, the parameterization of

[25] such as Monday vs. Tuesday, day vs. night, etc

those matrices is assumed to be generated by a uniform sampling over all such sets of parameters. As noted previously, if this assumption were violated, $p(M_c|I_0)$ in (1.17) would not be a constant. This section addresses the meaning of a uniform sampling of Markov matrices.

A Markov matrix with s states contains $s(s-1)$ independent parameters. Each row of the matrix contains $s-1$ independent parameters because any given entry can be determined uniquely by the constraint that the row sums to 1. The Markov assumption, which allows the distribution to be represented by a matrix, also implies that the rows of the matrix are independent probability distributions. In other words, knowledge of one row does not provide knowledge of another row. For this reason, uniformly sampling Markov matrices can be reduced to uniformly sampling potential rows and then combining them in an unbiased manner.

A row of a Markov matrix represents a discrete probability distribution and has the property that its entries are all non-negative and sum to one. The term *probability vector* is used to refer to such a row, and in general, this vector can be represented geometrically as a point on the $s-1$ dimensional simplex.[26] Each point on the simplex represents a unique probability vector. A uniform sampling of probability vectors corresponds to a uniform sampling of points on the simplex. The next section provides such an algorithm and proves its correctness.

1.9.1 A Naive algorithm for producing probability vectors

An obvious attempt at creating uniform probability vectors is to generate n random numbers at uniform between 0 and 1.[27] This vector can then be normalized by dividing each entry by the total vector sum. Unfortunately, such an algorithm produces a highly biased sampling of the simplex. Informally, the sum of n numbers generated by such a process is likely to be near $n/2$, which makes the entries generated after normalization more likely to be near $1/n$ than near 0 or 1. Interestingly, variations

[26]An n-degree simplex is embedded in $n+1$ dimensional space

[27]Such machinery is assumed to be present, and for the purposes of this section no distinction is made between random and pseudo-random numbers

of this algorithm are widely used, as evidenced by their frequent description in publications. The next section provides a superior algorithm and provides a proof of correctness based on differential topology.

1.9.2 Russell's Algorithm

Rather than simply normalizing a random vector, as described in the previous section, Russell's algorithm sorts a vector of length $n-1$ and considers the differences between successive elements. The vector of differences can be used as an unbiased sampling of the simplex, yielding the first $n-1$ components of the desired output. The final element is 1 minus the sum of the previous elements. Note that this algorithm produces a vector of length n by generating $n-1$ random numbers, reflecting the dependence of the final parameter upon the previous values.

Russell's algorithm [11] can be viewed as a mapping ϕ from points in the n-dimensional cube, $[0,1]^n$ to the n-dimensional simplex S_n. S_n is a subspace of $[0,1]^{n+1}$ with rank n. A probability measure on $[0,1]^n$ can be defined using the standard *Lebesgue* (volume) measure μ. This measure defines probability of an event E as the Lebesgue measure of E, normalized by the measure of the entire space $[0,1]^n$:

$$p(E) \equiv \frac{\mu(E)}{\mu([0,1]^n)} \tag{1.39}$$

The initial selection of n points at uniform on $[0,1]$ produces a uniform sampling of $[0-1]^n$.[28] This is true because the probability that a point is chosen in any hypercube F is equal to the ratio of the volume of F to the volume of $[0,1]^n$. However, as the first algorithm demonstrated, the fact that the algorithm samples uniformly from $[0,1]^n$ does not imply that it samples uniformly on S_n. For any region $F \in S_n$, μ can be used to define a probability just as in (1.39). In order for ϕ to be an unbiased sampling of S_n, the probability of the pre-image of F must be equal to the probability of F. In other words:

[28]This n is the previous $n-1$, but the -1 is dropped for convenience

$$p(F) \equiv \frac{\mu(F)}{\mu(S_n)} \overset{?}{=} p(\phi^{-1}(F)) \equiv \frac{\mu(\phi^{-1}(F))}{\mu([0,1]^n)} \tag{1.40}$$

The mapping ϕ can be decomposed into two steps: sorting the n numbers, and then mapping from the sorted list to S_n. Let T_n denote the space of all sorted vectors, as produced by the first step of Russell's algorithm. T_n can be viewed geometrically as the set of all skewed tetrahedrons in which the coordinates are sorted. The following definitions are used throughout the remainder of the proof:

$$[0,1]^n \equiv \{(x_1, ..., x_n) \mid \forall i, 0 \le x_i \le 1\} \tag{1.41}$$

$$T_n = \{(x_1, ..., x_n) \mid 0 \le x_1 \le x_2 \le ... \le x_n \le 1\} \tag{1.42}$$

$$S_n = \{(x_1, ..., x_n) \mid 0 \le x_i \le 1, \sum x_i = 1\} \tag{1.43}$$

$$\mu : \mathbb{R}^n \to \mathbb{R} \tag{1.44}$$

$$[0,1]^n \underbrace{\xrightarrow{f} T_n \xrightarrow{g}}_{\phi} S_n \tag{1.45}$$

The first step of the proof demonstrates that f is *measure preserving*. In other words, if E is a measurable subset of T_n, the measure of the pre-image of E must be proportional to the measure of E. Note that $[0,1]^n$ and T_n are related through permutations. Let P be the set of all permutations, then:

$$\bigcup_{\pi \in P} \pi T_n = [0,1]^n \tag{1.46}$$

If $\mathbf{x} \in \pi T_n$ then $f(\mathbf{x}) = \pi^{-1}\mathbf{x}$.[29] Note that for two distinct permutations π and π', the set $\pi T_n \cap \pi' T_n$ has measure 0, by Sarg's theorem, as it would require repeated elements prior to sorting.[30]

[29] π permutes T_n, therefore π^{-1} is the permutation that performed the sort.

[30] Repeated elements implies a condition of the form $x_i = x_j$ which is at most an $n-1$ dimensional subspace of $\mathbb{R}^n$

$$f(\mathbf{x}) \equiv \pi^{-1}\mathbf{x}, \text{for } \mathbf{x} \in \pi T_n \tag{1.47}$$

$$\forall \pi, \pi', \pi \neq \pi', \mu(\pi T_n \cap \pi' T_n) = 0 \tag{1.48}$$

For any measurable $E \subset T_n$:

$$\mu(f^{-1}(E)) = \sum_{\pi \in P} \mu(f^{-1}(E) \cap \pi T_n) = \sum_{\pi \in P} \mu(\pi E) = \sum_{\pi \in P} \mu(E) = (n!)\mu(E)$$

The second step uses the fact that $f^{-1}(\mathbf{x}) = \pi\mathbf{x}$. The third step uses the fact that $|\det(\pi)| = 1$. The final step uses the fact that $|P| = n!$. Now that f has been shown to be measure preserving, g is also shown to be measure preserving. Note that g can be written:

$$\begin{aligned} g_1 = & \quad x_1 \\ g_2 = & \quad x_2 - x_1 \\ & \vdots \\ g_n = & \quad x_n - x_{n-1} \\ g_{n+1} = & \quad 1 - x_n \end{aligned}$$

In order to show that g is measure preserving, it is sufficient to show that the *Jacobian* matrix, $|\frac{\partial g}{\partial x}|$ has a constant determinant. Unfortunately, attempting to write the *Jacobian* directly yields the following $(n+1) \times n$ matrix:

$$\mathbf{J_g} = \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ -1 & 1 & 0 & 0 & \dots & 0 \\ 0 & -1 & 1 & 0 & \dots & 0 \\ 0 & 0 & -1 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 & -1 \end{pmatrix}$$

This matrix is $(n+1) \times n$ and has 1's along the main diagonal and -1's along the sub-diagonal until the bottom row. The last row has a single -1 in the last position. Unfortunately this matrix is not square and hence has no determinant. However, as it has rank n, row and column operations could be performed to produce an upper $n \times n$ matrix and a lower row filled with 0's. Let κ be the determinant of that matrix. It is clear that this determinant does not depend on $\mathbf{x}$ as all of the entries are constants. By linearity:

$$\exists \kappa > 0, \forall F \in S_n, F \text{ measureable}, \mu(g^{-1}(F)) = \kappa \nu(F) \tag{1.49}$$

and referring back to (1.40):

$$p(F) = \frac{\nu(F)}{\nu(S_n)} = \frac{\mu(f^{-1}(g^{-1}(F)))\frac{n!}{\kappa}}{\mu([0,1]^n)\frac{n!}{\kappa}} = p(f^{-1}(g^{-1}(F))) = p(\phi^{-1}(F)) \tag{1.50}$$

as desired.

Chapter 2

Implementation

This chapter describes the implementation of an IDS using IBS. It focuses mainly on the theoretical aspects of its design. Portions that were physically implemented are highlighted throughout the chapter. It is organized as follows: first the architecture of an IDS and the role of IBS monitors are described. Second, the pre-processing steps necessary in order to run IBS are described. Third, the history of IBS and its development is detailed. Fourth, a general overview of the code written for this thesis is given, highlighting the changes and optimizations made, with a brief description of a side-project that aimed to parallelize the code. Finally, the machine learning aspects of the setup are described.

2.1 Building an IDS Using IBS

Many IDS are built by combining a set of data-monitors with a top-level monitor that interacts with the owner of a computer system in order to help the owner to prevent unauthorized activities. The Monitoring, Analysis, and Interpretation Tools Arsenal (MAITA) project [10] at the MIT Laboratory for Computer Science (LCS) provides a framework for integrating multiple monitors in order to construct such a system. A complete IDS would feature many different types of monitors, combining both anomaly and misuse detection, in order to best assess the likelihood that the system is behaving abnormally at any given time. IBS monitors built upon the algorithm described in the previous chapter would form a piece of such a suite. For this thesis, an IBS monitor was implemented that had full algorithmic functionality. However, it lacked many of the interfaces that would be necessary to fully integrate it into the MAITA framework. The design of such a monitor is now developed.

2.1.1 An IBS Monitor

As discussed in the previous chapter, sequences of system calls generated by privileged unix processes are believed to characterize the most error-prone operations of a computer system. For this reason, an IBS monitor would be designed to analyze the sequence generated by each such relevant process. Relevant processes would include those that are sufficiently complex and whose behavior depends upon actions made by agents. Such processes are generally server processes using the terminology of section (1.1.4). For each such process, an IBS monitor would be instantiated and the output of all such monitors would be used by the top-level monitor.

The appropriate level of sophistication for each IBS monitor can vary greatly depending upon various assumptions that must be made. In the crudest form, traces of system calls generated by a process could be used as input into IBS with the number of Markov states set equal to the number of system calls that the operating system allows. Unfortunately, as demonstrated in the experiments chapter, IBS performs far better with a number of states between five and 15 than it would with 250 states. This difference is due to the relationship between the influence of priors on estimated probabilities and the ratio of observed data to the number of parameters being estimated. For a very small number of states, IBS views all matrices as coming from different stochastic processes even when they should in fact be subsumed. For a very large number of states there is simply not enough data to fully characterize each matrix and they are all subsumed together, as priors dominate the information content. Therefore, a sensible pre-processing step would be to avoid allocating a Markov state for calls that are never used by a particular process. This form of pre-processing, as well as several others, is described in the next section.

Assuming that a suitable set of pre-processing rules has been determined, the next step in the setup of an IBS monitor would be the actual integration between the operating system and the monitor. In the most unsophisticated approach, system calls could be logged into a file that was then polled for changes and used to generate the time-series required by IBS. Unfortunately, this method introduces a sizeable lag between the actual system call and the appearance of that data-point in the time-series (and ironically, would involve dozens if not hundreds of system calls made by other processes). Also, the system call logging software is generally optimized for performance rather than reliability and is known to drop system calls from the log on a fairly regular basis. Finally, although the logging software attempts to correctly interleave the system calls made by child processes of the server process, the logging software again favors performance over reliability and does not always produce a true ordering of events. For these reasons, a more sophisticated monitor should be integrated directly into the kernel. For the purposes of this thesis, data was collected via standard system-call logging utilities such as truss for the Sun operating system.

Along with pre-processing decisions, the appropriate priors and β must also be determined. These parameters must be communicated from the IDS to the individual monitors. Similarly, the output of each monitor must be interpreted by the IDS top-level monitor. Fortunately, the communications necessary to perform these tasks are generally handled by a generic framework such as the previously mentioned MAITA project. The remainder of the thesis ignores the communication difficulties that arise when trying to coordinate multiple asynchronous processes and assumes that the IBS monitors are able to easily communicate with the top-level of the IDS.

The output of an IBS monitor is viewed as a time-series of cluster membership (along with the implied mapping from cluster identifiers to the time at which the cluster was first created). In other words, at each point in time IBS produces a value that classifies the recent state of the system as being in a particular cluster. This value is piece-wise constant, changing only at break-points. The library of clusters grows over time but should eventually reach a steady-state in which the vast majority of system behavior agrees with previously observed patterns. While this is not true in the most general case, it is likely to be true for computer programs that are intentionally designed to behave systematically. The most interesting events are the creation of new clusters. This occurs when the system behaves in a manner that cannot be explained by previously observed behaviors. Observing such an event should increase the top-level monitor's belief that the system is behaving abnormally. Similarly, observing the subsumption of a new segment into a cluster labeled as abnormal or into a rarely used cluster indicate interesting events to the IDS. In general, the top-level of an IDS uses multivariate analysis of each of its monitors in order to make decisions regarding how to react to changes in system state. Such decisions are an active research topic by themselves but are ignored for the remainder of this thesis. The actual implementation had no official top-level monitor and was run in an interactive manner, requiring the user to type commands into a terminal. However, the level of interaction required during operation was minimal and could be automated using pre-recorded files as input.

One interesting feature of combining IBS monitors with a top-level monitor is the possibility for feedback. For example, parameters such as β and the usage of priors could evolve dynamically over time. In general, the role of the top-level monitor is to tune the system's overall specificity and accuracy. Adjusting β is one means of making the system more or less likely to report anomalous activity as it determines how quickly break-points are detected after system behavior appears to change. The experiments chapter explores the relationship between β and the overall accuracy and specificity of IBS. The top-level monitor is also in a position to trade-off CPU usage versus monitor accuracy. For instance, when it notices that the system has idle cycles, it could direct an IBS monitor to optimize itself by running an iteration of BCD clustering or pruning out rarely used clusters.[1] In general, the IDS provides a gateway for the owner of the system to change the parameters of IBS either directly through human intervention or through an AI interface.

One additional possibility for feedback between the IDS and IBS is the addition of new matrices based upon information that surfaces in the real-world after the analysis

[1]As discussed in the derivation, such clusters may have resulted from incorrectly combining segments during the initial few subsumption tests.

has begun. For example, the discovery of new attacks may warrant instructing IBS to add new matrices to its library of clusters. In addition, the creation of new system calls or the deprecation of older calls may necessitate changes to the internal data-structures. Finally, if the IDS were integrated with the kernel, it may be feasible to halt the operations of a suspicious process pending a more complete analysis of its recent behavior by a set of monitors. In the case of IBS it may be possible to re-run analysis on a recent subset of the time-series using BCD clustering in order to double-check the decisions that were made regarding its behavior.

2.2 Pre-processing Implementation

As described in previous sections, a great deal of data pre-processing is required before IBS can be run. First, the file format produced by the system call logging utility must be transformed into a suitable format for reading by IBS. Second, the system calls must be encoded from their original format onto the integers between 1 and N. The proper choice of N must be made in a manner that is both backwards and forwards compatible (A commercial application would most likely need to deal with updating the learned data when new system calls were adopted by the operating system). Also, if N is determined by scanning a finite length sequence of system calls, it is possible that it does not contain system calls sufficiently rare that they never occur in the sequence. In such cases it would be most appropriate to view the pre-processing as a particularly naive form of IPS in which it triggers an alert whenever it encounters a system call that it cannot map. In order to motivate the need for additional forms of pre-processing, actual data collected from the `sendmail` process is analyzed.

Figure 2-1 shows a chart of the system calls made by sendmail.[2] The entries are sorted by the number of times that each call was made. The cumulative column shows the cumulative density function (CDF) beginning with the first entry. The entropy column shows the entropy of the corresponding row in a maximum-likelihood

[2]This data-set was collected by the University of New Mexico and is available at `http://www.cs.unm.edu/~immsec/papers.htm`

transition probability matrix generated from this data. Lower entropy indicates that observing the system call gives a better prediction for the next system call, and conversely, creates a larger surprise when the next system call is not consistent with previously observed data. First, note that 30% of calls are made by the most common call, 50% by the top two, 96% by the top six and 99% by the top seven. Also note that the entropy increases dramatically after the 7^{th} call and again after the 19^{th} call. Finally, note that sendmail uses 48 of about 250 system calls but that only 36 of them occur more than 10 times. As described in the previous section, system calls like `setpgrp`, `getpagesize`, and `getrlimit` should never appear after sendmail has started servicing client requests and should therefore be detected using an IPS rather than including them in the list of Markov states used by IBS.

Several important conclusions are drawn from this analysis. First, the number of times that each system call is made and how informative it is about the next system call range very widely. Second, the process does not use all available system calls and of the calls that it uses a significant proportion are called only several times. Next, system calls that are sufficiently rare provide very little predictive power for the next system call, as the row estimate is completely dominated by the prior. Fortunately, such system calls can easily be detected using a much less sophisticated IPS (and in fact, can be detected while performing the necessary pre-processing). Finally, although not evident in this particular histogram, many different system calls that are functionally similar, such as `open` and `open64`, are used by the same application in different portions of the code. Given that an attacker could choose to use either `open` or `open64`, it seems prudent to collapse them into the same Markov state rather than relying on him choosing to favor one method over the other. These facts suggest that pre-processing should be used to achieve some level of state-reduction before calling IBS. Therefore, the goal of pre-processing is to reduce the set of all possible system calls into a set of pseudo-calls that best characterize the behavior of a process.

For this thesis, pre-processing re-classified the 48 system calls in figure (2-1) into 13 pseudo-states representing the top 10 system calls and 3 pseudo-calls referred to as `rare1`, `rare2`, and `rare3`. The mapping was performed by hand, based upon points

```
rank    calls   name    cumulative      entropy
1       158474  sigvec          0.3186  1.00573117772406
2       95139   sigblock        0.5098  0.661528809757723
3       79250   sigsetmask      0.6692  1.41357321988662
4       47994   getpid          0.7656  1.02059643778272
5       47874   gettimeofday    0.8619  0.963031265149759
6       47495   setitimer       0.9574  0.996698250292566
7       15889   sigstack        0.9893  0.0806804236458789
8       1253    close           0.9918  2.75063890220242
9       560     read            0.9929  2.64182447937747
10      465     open            0.9939  2.98540336365839
11      427     write           0.9947  2.67895483583741
12      415     socket          0.9956  1.97804136640904
13      404     ioctl           0.9964  1.98160610920957
14      244     sendto          0.9969  2.24142212675598
15      211     fstat           0.9973  2.88029625284931
16      207     bind            0.9977  1.7412091223398
17      206     recvfrom        0.9981  1.71879570959752
18      206     select          0.9985  1.71879570959752
19      206     connect         0.9990  1.71879570959752
20      72      unlink          0.9991  4.17018857760515
21      52      wait4           0.9992  4.32594971174418
22      44      lseek           0.9993  4.23980416979795
23      39      fork            0.9994  4.43062123711694
24      38      dup             0.9994  4.66446938748727
25      36      stat            0.9995  4.6742561250704
26      36      chmod           0.9996  4.6742561250704
27      27      accept          0.9996  4.82336955390068
28      25      getuid          0.9997  4.80066448703563
29      25      link            0.9997  4.8444624674873
30      24      access          0.9998  4.8336551032247
31      14      getdtablesize   0.9998  5.11378153077665
32      13      getgid          0.9998  5.10933215172954
33      12      pipe            0.9999  5.10512865667795
34      12      creat           0.9999  5.10512865667795
35      12      vfork           0.9999  5.10512865667795
36      12      rename          0.9999  5.10512865667795
37      7       mmap            1.0000  5.4767253953035
38      6       getdents        1.0000  5.481648844716
39      3       fcntl           1.0000  5.53997656741928
40      2       setsockopt      1.0000  5.56385618977473
41      2       getdomainname   1.0000  5.56385618977473
42      1       setpgrp         1.0000  5.5738935175846
43      1       getpagesize     1.0000  5.5738935175846
44      1       getrlimit       1.0000  5.5738935175846
```

```
45      1       listen          1.0000  5.5738935175846
46      1       chdir           1.0000  5.5738935175846
47      1       umask           1.0000  5.5738935175846
48      1       gethostname     1.0000  5.5738935175846
```

Figure 2-1: System calls made by sendmail during normal operation

at which the entropy changed dramatically. A number of automated approaches were considered but fell outside of the possible scope of the thesis.

Several other pre-processing steps were used. Although not relevant for sendmail, any system call ending with a 64 was equated with the non-64-bit version. Points at which the process identifier (PID) of the system calls changed were annotated in the data-set. Such changes occur when the operating system chooses to preempt one child (forked) process in favor of another (preempting to an unrelated process would not be visible in the trace). The IBS implementation allowed for such annotations to be treated as forced break-points with the understanding that this would be roughly equivalent to performing a multivariate analysis in which one stream of data consisted entirely of PIDs. Also, for training data, the interleaving of child processes was undone by collating the calls made by each child process. In other words, the data-set was re-ordered as if each child process ran to completion before another was allowed to run. This step was made to avoid contaminating the data for each child process with the first few system calls made by the next process each time the operating system performed a context switch. Finally, it was observed that traces of system calls often contain long series in which the same call repeats many times. For example, many unix processes attempt to close every possible file-descriptor at various stages in the code in order to guarantee that no descriptors can be inherited when they fork. Similarly, a long string of `read` or `write` system calls could indicate reading or writing a large file. In these cases, it would be most natural to assign a label to the behavior that is not the same as the label of an individual system call. As an approximation to modeling this behavior, experiments were performed in which all such series were reduced to contain a single element. Such experiments performed consistently worse than their counterparts and this form of pre-processing was abandoned.

The following additional types of pre-processing were considered but not implemented:

1. Standard state reduction techniques such as singular value decomposition or principal component analysis

2. State reduction using grammar learning to equate common patterns of system calls with a single pseudo-state

3. Working with domain experts to create synthetic matrices that would be indicative of an attack

4. State reduction using cross entropy between normal and attack traces (removing system calls that were not informative about whether the system was in a normal or attack state for a pre-specified list of known attacks)

2.2.1 Parsing truss files

Perl code was written to parse truss files generated by the standard SunOS system call logging utility. Although not a part of IBS itself, these scripts are an important part of the IDS. In a full implementation they would most likely be moved inside of the main parsing loop for efficiency. The following options are currently available:

1. System calls present in a set of logs are mapped from their original encoding on [1,250] to [1,N] where N is determined empirically.

2. System calls ending with 64 can optionally be treated as equivalent to the original version.

3. Changes between PID can be annotated in the file. In addition, the sequences for each PID can be collated. For training data, it is reasonable to perform collation to diminish the noise introduced by the operating system scheduler. This collation is not performed after the training phase.

4. State reduction through explicit mapping, e.g. treating fork and rfork as equivalent.

5. Optionally collapsing repeated system calls into a single call, for example mapping `close,close,close,...,close` to a single `close`.

2.2.2 Pre-loading the library of Matrices

As discussed in the derivation of IBS, the initial subsumptions are likely to contaminate the true probability distributions as they do not contain enough information to correctly classify the segments. Given an infinite amount of data, these initial mistakes would not affect the final matrices. However, for any finite time-series there is incentive to avoid such mistakes. One means of avoiding these initial misclassifications is to pre-load the library of clusters with pre-created matrices known to match patterns of system behavior (either normal or abnormal). These matrices could be created as a side-effect of the pre-processing steps described in the previous section, or by other means. One method of obtaining a set of matrices would be to perform IBS segmentation to obtain a set of segments and then to use BCD rather than IBS clustering. The BCD clustering, which is much more computationally expensive, would result in a better set of clusters than would be created by IBS's incremental approach.

2.3 Project History

The IDS built for this thesis is based upon code written in common LISP by Marco Ramoni in 1998. His *Dyncoscope* program implemented IBS as discussed in the derivation, using the standard clustering algorithm and taking a single argument specifying the equivalent sample size. Unfortunately, the LISP code pre-scanned the entire dataset in order to determine the total number of discrete Markov states. In addition, the LISP parsing code created a monolithic list in memory. This method did not work when applied to the IDS data sets as they were too large to be read into memory prior to processing. For this reason, the code was ported to java and changed to take the number of states as an additional parameter.

The java version of the code was originally written as a nearly exact duplicate of the LISP code, with no attempts made to optimize the code for java-specific features. However, it was able to process much larger data sets than the LISP version as it read through the data-set incrementally rather than pre-scanning the entire file into memory. After the java code was carefully verified to produce duplicate output, experimental changes and optimizations were made in order to pursue the experiments performed in this thesis.

Although the java version was able to read much larger data-sets than the LISP version, it was still extremely slow. Training on a 500,000 transition data-set took approximately 10 hours. Upon a closer profiling of the code, it was determined that the algorithm exhibited a high degree of parallelism and an experiment was performed to determine if the algorithm could be effectively parallelized using MPI, the standard super-computing communication scheme. In order to perform the experiment, the code was ported again, from java to C++, as the MPI libraries were C/C++ compatible. This port had the added benefit of making the execution considerably faster, even when running on a single processor. Finally, the C++ version was ported to pure C in order to facilitate testing using the Cilk parallelization toolkit. Cilk is an experimental project at the MIT Laboratory for Computer Science. Unfortunately its compiler supported only C at the time when this experiment was performed. The MPI version of the parallelized code outperformed Cilk but the final version of the IBS code is written in C.

2.4 Code Overview

The C version of IBS is written as a single executable that takes a number of command-line arguments discussed in the following section. It is broken into three source files: ibcd.h, ibcd.c and ibcd_main.c. The main entry point of the code, located in ibcd_main.c, parses the arguments and enters the file parsing loop. At this time, the data-set is read sequentially from a single file, although in principle the code could easily be adapted to read from a network device or through other means. Each transition causes the count matrix, known in the code as a `process` struct, to be updated in a function called `update_current_process`. Next, a function known as `update_current_score` is used to perform the break-point detection as described in section (1.7.3). Finally, the function `process_under_control` determines if a break-point has been detected. Relevant portions of the code are shown below with error-checking and other non-algorithmic sections removed:

```
typedef struct process {
  int position;
  bool type;
  int frequencies[STATES][STATES+1];
  double probabilities[STATES][STATES];
  int transitions;
  double likelihood;
} process;
```

This data structure stores a count matrix and the associated transition probabilities. It also records the type of process in the field `type`. Currently this is a boolean indicating only normal or abnormal. However, future implementations may have more than two values for this variable. The matrix `frequencies`, implemented as a doubly-referenced array, stores an extra element in each row. The extra element stores the row sum, which is used to calculate probabilities. This tradeoff between space and time was clearly favorable after initial testing. The variable `likelihood` is used to cache the value of the likelihood score for this matrix. This optimization is discussed in the following section.

```
void update_current_process(const int prev,
                            const int val,
                            process* proc) {
  proc->frequencies[prev][val]++;
  proc->frequencies[prev][STATES]++;
  proc->transitions++;
}
```

This function updates the count matrix after a transition. It increases the row count as well as the total number of transitions.

```
void update_current_score(const int prev,
                          const int val,
                          const process* proc) {
  double score = (-log(proc->probabilities[prev][val]));
  double mean = 0;
  double variance = 0;
  double c = 0;
  double diff;
  for(i=0; i<STATES; i++) {
    c = proc->probabilities[prev][i];
    mean += (c * -1 * log(c));
  }
  for(i=0; i<STATES; i++) {
    c = proc->probabilities[prev][i];
    diff = (-1 * log(c)) - mean;
    variance += diff * diff * c;
  }
  v.score += score;
  v.mean += mean;
  v.variance += variance;
}
```

This function performs the segmentation described in (1.7.3). The global data structure v stores the $\texttt{score}_t$, $\texttt{mean}_t$, and $\texttt{variance}_t$ variables.

```
bool process_under_control() {
  double sd = sqrt(v.variance);
  double margin = sd * ibcd.cutoff_shold;
  double lower = v.mean - margin;
  double upper = v.mean + margin;

  if(v.score <= upper &&
     v.score >= lower) {
    return true;
  }
  return false;
}
```

This function checks whether the difference between $\texttt{score}_t$ and $\texttt{mean}_t$ has exceeded the required threshold for a break-point. Each time that the main parse loop detects a break-point, the current count matrix is passed to the clustering portion of the algorithm. The clustering is performed by a function known as `check_out_process`. This function computes the likelihood of the model in which the new process is added to the library of matrices as a new entry. It then computes the likelihood of each possible subsumption of the new segment into an existing cluster. It chooses the best possible model and updates the library accordingly. In addition, command-line arguments can be used to specify that MDL clustering be performed rather than the standard likelihood clustering. Both the Bayesian approach and the MDL approach take advantage of a function called `compute_marginal_likelihood`, which returns the score of a library of matrices. A helper function known as `compute_subsumed_marginal_likelihood` handles the temporary modifications necessary to alter the library for calculating the score of a given subsumption. Therefore, `check_out_process` is written as a loop around calls to `compute_subsumed_marginal_likelihood`, which in turn calls `compute_marginal_likelihood` to loop over the library. Fortunately, this n^2 operation can be made a linear operation by caching the likelihood scores for each process.

```
process* check_out_process(process* proc,
                           const int start,
                           const int end,
                           process_list* processes) {
  double best_score = 0.0;
  const int stored_processes = processes->length;
  process* best_cluster = NULL;
  process* candidate = NULL;
  int i;
  process* cluster = NULL;
  process* subsumed_result = NULL;
  double score = 0.0;

  // temporarily add this process to library to get the score
  addHead(processes,proc);
  best_score = compute_marginal_likelihood(processes);
  removeHead(processes,proc);
  // now calculate each subsumed score

  for(i=0; i<stored_processes; i++) {
    cluster = get(processes,i);
    score = compute_subsumed_marginal_likelihood(proc,
                                                 cluster,
                                                 &subsumed_result,
                                                 processes);
    if(score >= best_score) {
      best_score = score;
      best_cluster = subsumed_result;
      candidate = cluster;
    }
  }

  if(best_cluster == NULL) {
    return store_new_process(proc,processes);
  }
  else {
    return store_subsumed_process(proc,
                                  candidate,
                                  best_cluster,
                                  processes);
  }
}
```

```
double compute_subsumed_marginal_likelihood(process* proc,
                                            process* cluster,
                                            process** subsumed,
                                            process_list* processes) {

  int i,j;
  double score;
  *subsumed = make_process2(cluster->type,
                            cluster->index);

  // combine the two count matrices
  for(i=0; i<STATES; i++) {
    for(j=0; j<=STATES; j++) {
      (*subsumed)->frequencies[i][j] =
        proc->frequencies[i][j] +
        cluster->frequencies[i][j];
    }
  }

  (*subsumed)->transitions = proc->transitions + cluster->transitions;

  // switch the new matrix into the library
  replace(processes,cluster,*subsumed);
  score = compute_marginal_likelihood(processes);
  // replace the original matrix back into the library
  replace(processes,*subsumed,cluster);
  return score;
}
```

The code for `compute_marginal_liklihood` is omitted for brevity but follows directly from (1.26).

2.5 Command line arguments

The free parameters for the algorithm are listed below. At the present time some of these options are available only through re-compilation rather than as actual command line arguments.

1. The number of states: s
2. Whether forced break-points are allowed (based upon data-set annotation)

3. Whether only forced break-points are allowed (disables normal segmentation)
4. The break-point threshold: β
5. The value of α_{kij}, either a constant or loaded as a matrix
6. The type of clustering: Bayesian or MDL (the degrees of freedom are also variable)

2.6 Optimization

Several major optimizations were made to the IBS code that were not described in the original paper[21]. First, the count matrices were extended to contain row totals. This tradeoff between space and time was clearly optimal in the C version. Second, the struct `process` storing the count matrix also caches the value of its transition probabilities and log likelihood. These values are discarded whenever the matrix is permanently updated, which occurs only in the temporary matrix used for segmentation or when a subsumption occurs. Finally, many memory oriented optimizations were made using advanced pointer techniques such as reference counting and copy-on-write. In the single-processor version, a single matrix library is used and pointer manipulations are used in order to avoid copying the count-matrices in order to test a subsumption. This implementation uses far less memory than the java or LISP versions which required copying the count matrices each time a subsumption was tested. The parallelization optimization is discussed in section (2.9).

2.7 Training vs Classification

An important factor in the utility of an IDS is the amount of required interaction with the owner of the computer system. In order to distinguish between normal and abnormal behavior, some decision has to be made regarding the interpretation of the system's output. In IBS several interesting types of events occur. First, the creation of a novel cluster indicates that the time-series contained a set of transitions unlike any previously observed. Second, the subsumption of a new segment into an existing cluster that has relatively few transitions is also a rare event. If the target cluster is sufficiently rare compared to the total length of the sequence, this event may also indicate abnormal system behavior. Implicit in the interpretation of these events is a distinction between the training and classification phases of the IDS. A special flag, the `type` field in the `process` struct, is used to indicate whether a cluster is considered normal or abnormal. By default this value is set to normal for all clusters created during training and to abnormal otherwise.

The training phase of IBS is the initial period of its execution. During this time, the data may receive extra pre-processing, as by definition, this is the phase in which human interaction is feasible. This phase may also use a carefully controlled or synthetic data-set rather than sampling directly from the oracle. Furthermore, this phase, as it is of finite length, may consume more time per transition than the eventual $O(1)$ time required to analyze an infinite sequence. The goal of the training phase is to tune IBS to most effectively detect interesting events during the classification phase. In the current implementation the distinction between training and classification is annotated in the time-series.

The beginning of the classification phase is the point at which the goal of the IDS shifts towards avoiding human interaction while detecting interesting events. The key measures of performance are the sensitivity and specificity of the classification of segments as normal or abnormal. Unfortunately, as IBS classifies segments of transitions rather than individual transitions, the standard metrics are difficult to apply. In general, the goal of the IDS is to detect a break-point as quickly as possible following the beginning of an attack and to minimize the number of segments and transitions that are incorrectly labeled as abnormal. Unfortunately there are many ways in which a sequence can be abnormal. The IDS must provide a means for the owner to specify if an abnormal sequence should, in the future, be considered abnormal, or it should be treated as a rare but expected feature. In general, an IDS suite consists of many algorithms like IBS and uses higher-order algorithms to interpret their results. However, this thesis implements only the IBS portion of the suite.

2.8 Priors

Several choices exist for defining the priors used in IBS. First, an (approximately) uninformative prior of $\text{Dir}(1, 1, ..., 1)$ could be used at every point of the process. Without making any assumptions about the types of processes likely to be observed, this is the logical choice. However in an actual IDS it may be feasible to make further assumptions. The most basic assumption would be that processes generally follow the global histogram of all system calls across all computer processes. Such a histogram could easily be obtained by recording every system call made by a computer for some arbitrary length of time sufficient to produce an estimate of suitable accuracy. Another related assumption would be that behaviors vary from computer program to computer program, and that the histogram obtained from the specific computer process should be used. Finally, sophisticated analysis of the computer code for a process could be used to determine a set of suitable priors. Such analysis could either be performed by hand or by scanning the executable for system calls. In certain domains, experts may be able to provide suitable estimates for the prior probabilities. In the case of computer systems, certain system calls are much more dangerous than others, and this level of importance could be used to bias the priors towards responding more quickly to changes in dangerous system calls. Finally, hybrid methods exist in which an uninformative prior is gradually mixed with the observed histogram. The IBS implementation developed for this thesis uses the uniform priors approach. However, it supports reading the priors from files that could be generated using any of the methods listed above.

2.9 Parallel MPI Version of IBS

The C/C++ code for IBS was also ported to support MPI parallelization of the algorithm. This effort was not a focus of the thesis, but it did provide a means of reducing computational time for performing training.

2.9.1 Code Modifications

Three major code modifications were necessary to write an MPI version of IBS. First, marshaling code was written in order to broadcast a segment from the root node to each of the other nodes. Second, the main loop was modified so that the root node parsed the file and performed the break-point detection while each of the other nodes waited to be sent a segment. Finally, `compute_subsumed_marginal_likelihood` was modified to perform only a subset of the computations based upon the rank of the processor. At the end of this function, `MPI_Allreduce` was used to calculate and distribute the globally optimal index of the matrix subsumption.

Marshalling

It was necessary to write functions to broadcast a count matrix between the root node and all of the other nodes. Portions of this code are shown below:

```
#define BROADCAST_SIZE 7600
int broadcast_buffer[BROADCAST_SIZE];

typedef struct broadcast {
  int start;
  int end;
  process* proc;
  bool done;
} broadcast;

void broadcast_segment(process* proc,
       const int start,
       const int end,
       bool done) {
  int i,j,k;
  broadcast_buffer[0] = start;
  broadcast_buffer[1] = end;
  broadcast_buffer[2] = proc->index;
  broadcast_buffer[3] = proc->position;
  broadcast_buffer[4] = proc->type;
  ...
  MPI_Bcast(broadcast_buffer,BROADCAST_SIZE,MPI_INT,0,MPI_COMM_WORLD);
}

broadcast* receive_broadcast(int r) {
  int i,j,k;
  process* proc = make_process2(-1,-1);
  if(rank == 0) {
    die(__LINE__,__FILE__);
  }
  MPI_Bcast(broadcast_buffer,BROADCAST_SIZE,MPI_INT,0,MPI_COMM_WORLD);
  bc.start = broadcast_buffer[0];
  bc.end = broadcast_buffer[1];
  bc.proc = proc;
  bc.done = broadcast_buffer[k++];
  return &bc;
}
```

Main Control Loop

The outer main loop was altered slightly such that only the root node began parsing the input file. The other nodes entered an infinite loop where they waited to receive broadcasts in order to do subsumption work. They exited the loop when a flag was set in the broadcast structure.

```
  if(rank == 0) {
    db = fopen(argv[1],"r");
    ...
  }
  else {
     while(true) {
      pBC = receive_broadcast(rank * 17);
      ...
     }
  }
```

check_out_process

`check_out_process` performs the segment classification and was re-written slightly to perform the broadcast and then calculate only a subset of the subsumption scores.

MPI_Allreduce was then used to find the global optimum and the relevant changes were made to the library of matrices.

```
broadcast_segment(proc, start, end, done);
local_score.score = compute_marginal_likelihood(processes);
local_score.index = -1;
for(i=0; i<stored_processes; i++) {
  if(i % np != rank)
    continue;
  score = compute_subsumed_marginal_likelihood(proc,
                                               get(processes,i),
                                               &subsumed_result,
                                               processes);
  if(score >= local_score.score) {
    local_score.score = score;
    local_score.index = i;
    best_cluster = subsumed_result;
    candidate = cluster;
  }
}
...
MPI_Allreduce(&local_score,
              &global_score,
              1,
              MPI_DOUBLE_INT,
              MPI_MAXLOC,
              MPI_COMM_WORLD);
if(global_score.index == -1) {
  ret = store_new_process(proc,processes);
}
else {
    candidate = get(processes,global_score.index);
    ret = store_subsumed_process(proc,
                                 candidate,
                                 best_cluster,
                                 processes);
}
...
```

Similar code appeared in the branch of the main loop executed on the non-root nodes. In this manner the subsumption was performed in parallel after the broadcast of the latest segment. Each node performed the same actions on its local library of matrices and the result was a distributed and replicated set of matrices.

2.10 MPI Performance Results

The MPI version of IBS produced a linear speedup when the first additional processor was added. However for more than 2 processors, while there was a slight speedup, the effect was negligible. This is probably due to the fact that the broadcast between the first two processors occurred on the same machine while adding more processors required using TCP/IP. Also, the marshaling code was highly un-optimized and required sending 7600 bytes. More tuning could have been performed to lower this number by sending only the count matrices and re-generating the associated probabilities at each node. However, for the purposes of this thesis a 2x speedup was sufficient. Figure 2-2 shows the contents of the data-sets while figure 2-3 shows the results.

File	Matrices	States	Execution length	Cycles	Transitions
50.lisp	16	50	100	20	32,000
100.lisp	32	50	100	20	64,000
150.lisp	64	50	1000	40	2,560,000

Figure 2-2: The `50.lisp`, `100.lisp`, and `150.lisp` data-sets

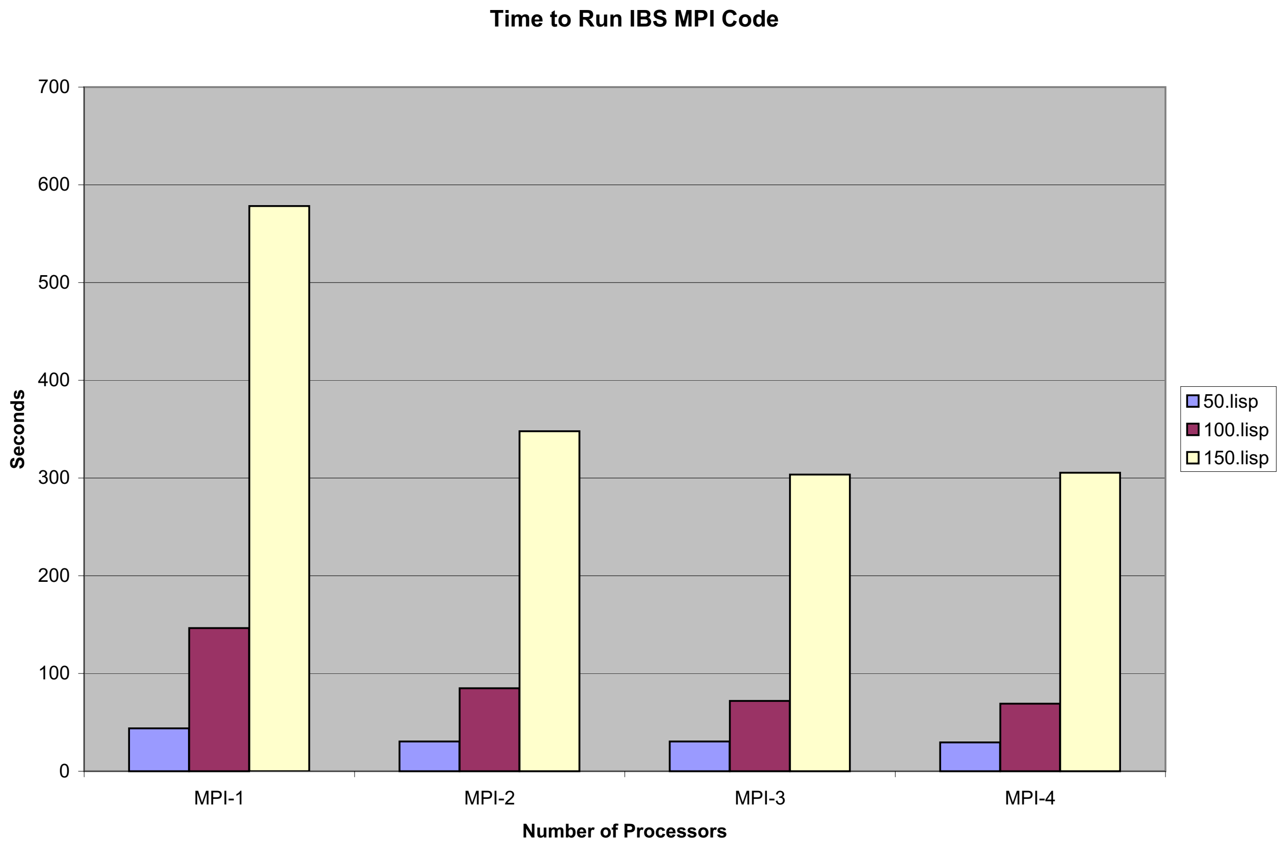

Figure 2-3: Elapsed time of IBS MPI code run on 1-4 processors

Chapter 3

Experiments

This chapter describes experiments that were performed in order to assess the effectiveness of IBS as part of an IDS. Three major types of experiments were performed: exploring the relationship between β and other parameters, measuring clustering accuracy, and testing the effectiveness of library pre-loading. In addition, several tests were performed to analyze the sendmail traces used to produce the histogram in the previous section. The analysis of sendmail is presented first, followed by the β experiments, the clustering experiments, and the pre-loading experiments.

3.1 Analyzing Sendmail

The sendmail program is one of the most popular targets of computer crackers due to its enormous complexity and wide installation base. It serves two basic purposes: delivering email to a local user's mailbox, and forwarding mail that doesn't belong to a local user to another mail server. In order to deliver mail to a user's mailbox it must be able to assume the identity of that user, and therefore it must run as a privileged process. It also performs other privileged actions such as listening to privileged networking ports and authenticating local users. Figure (2-1) in the previous chapter detailed a histogram of system calls made by sendmail. The following histograms show similar data for traces collected on a machine at MIT (medg.lcs.mit.edu):

The observation that different child processes exhibit very different histograms is

```
fid     count   %calls  CDF     fname
12      1960    21%     21%     time
224     1391    15%     37%     xgetid
5       1089    12%     49%     close
217     956     10%     59%     xctl
222     932     10%     69%     xstatvfs
143     932     10%     79%     poll
219     776     8%      88%     xwait
220     387     4%      92%     xfork
92      363     4%      96%     accept
38      363     4%      100%    pipe
```

Figure 3-1: System calls made by the main sendmail process

```
fid     count   %calls  CDF     fname
217     30      19%     19%     xctl
216     23      14%     33%     xstat
224     22      14%     47%     xgetid
2       16      10%     57%     read
3       12      8%      64%     write
4       11      7%      71%     open
218     9       6%      77%     xseek
5       9       6%      82%     close
12      8       5%      88%     time
111     6       4%      91%     getsockopt
98      4       2%      94%     setsockopt
223     2       1%      95%     xconnect
220     2       1%      96%     xfork
143     2       1%      98%     poll
225     1       1%      98%     xsetuid
222     1       1%      99%     xstatvfs
206     1       1%      99%     _exit
9       1       1%      100%    unlink
```

Figure 3-2: System calls made by a sendmail child process

fid	count	%calls	CDF	fname
216	99	25%	25%	xstat
217	70	17%	42%	xctl
224	49	12%	54%	xgetid
5	38	9%	63%	close
4	27	7%	70%	open
2	27	7%	77%	read
218	19	5%	81%	xseek
12	16	4%	85%	time
3	12	3%	88%	write
143	6	1%	90%	poll
111	6	1%	91%	getsockopt
225	5	1%	93%	xsetuid
98	4	1%	94%	setsockopt
41	4	1%	95%	setgid
63	3	1%	95%	mmap
9	3	1%	96%	unlink
223	2	0%	97%	xconnect
220	2	0%	97%	xfork
186	2	0%	98%	sysconfig
72	2	0%	98%	setgroups
65	2	0%	99%	munmap
219	1	0%	99%	xwait
206	1	0%	99%	_exit
190	1	0%	99%	sysinfo
135	1	0%	100%	getrlimit
121	1	0%	100%	rename
38	1	0%	100%	pipe

Figure 3-3: System calls made by a second sendmail child process

```
fid     count   %calls  CDF     fname
216     64      31%     31%     xstat
224     48      23%     55%     xgetid
217     25      12%     67%     xctl
5       18      9%      76%     close
4       13      6%      82%     open
218     8       4%      86%     xseek
12      8       4%      90%     time
2       8       4%      94%     read
3       3       1%      95%     write
223     2       1%      96%     xconnect
225     1       0%      97%     xsetuid
220     1       0%      97%     xfork
206     1       0%      98%     _exit
140     1       0%      98%     getsockname
132     1       0%      99%     getpeername
111     1       0%      99%     getsockopt
72      1       0%      100%    setgroups
35      1       0%      100%    kill
```

Figure 3-4: System calls made by a third sendmail child process

favorable for IBS, as it shows that the Markovian model of sendmail is justified. The first histogram shows the top-level process while the other histograms show several forked processes. A quick experiment showed that IBS was able to form separate clusters for each of the processes above when the data was collated, but not when it was interleaved. Given that these processes each represent functionally different pieces of the code, this result inspired the collation of traces by PID during pre-processing of training data.

The next aspect of sendmail that was analyzed was the distribution of KL distances between the clusters generated by each of the separate processes. A histogram of such distances is shown in figure (3-6). A similar figure is shown for matrices generated using Russell's algorithm in figure (3-5). Several conclusions can be drawn from these figures. First, the sendmail histogram shows that sendmail processes exhibit a wide range of KL distances, with no obvious concentration of mass in the histogram. The figure for randomized processes is very different with a large concentration around 1. Theoretical analysis can be used to show that the expected value of the KL divergence between two random vectors in n-space is $\frac{n}{n+1}$, which is verified by this figure, as the curves for matrices with a higher number of states are more concentrated at 1. Unfortunately, these figures show that the assumption that the matrices chosen by the oracle are a random sample of all possible matrices is very inaccurate for sendmail. Instead, there are many similar matrices as well as many matrices that are extremely distant from one-another in KL space. This result means that the IBS segmentation is based upon a faulty assumption, although the data does not suggest any particular changes to make. Nevertheless, these observation inspired adding an "optimal" category to the experiments performed later in this chapter in which segmentation was performed based upon data-set annotations rather than using IBS segmentation. These "optimal" results show a theoretical upper-bound on performance that could be achieved by IBS.

Finally, the distribution of the lengths of executions was analyzed. Figures (3-7) and (3-8) show the distribution of the number of system calls (transitions) as well as the number of unique calls made per process. The first figure shows that most executions are around 200 system calls. On average, each process was interrupted about three to four times, for an average execution length between 50 and 100 transitions. The second figure shows that the vast majority of processes use around 20 system calls. These two figures, along with the previous observations, were used to choose values of 10, 50, 100, and 1000 for test execution lengths and 6, 12, and 18 for the number of states. 10 was chosen as a minimum length to demonstrate the debilitating effects of switching processes before enough data has been collected, while 1000 was chosen to establish an upper-bound on performance. Similarly, 6 was chosen as a lower bound on the amount of state reduction that could be performed, while 12 and 18 represent reasonable Markov representation of the behavior of sendmail. The remainder of this chapter uses synthetic data sets that have properties inspired by this analysis.

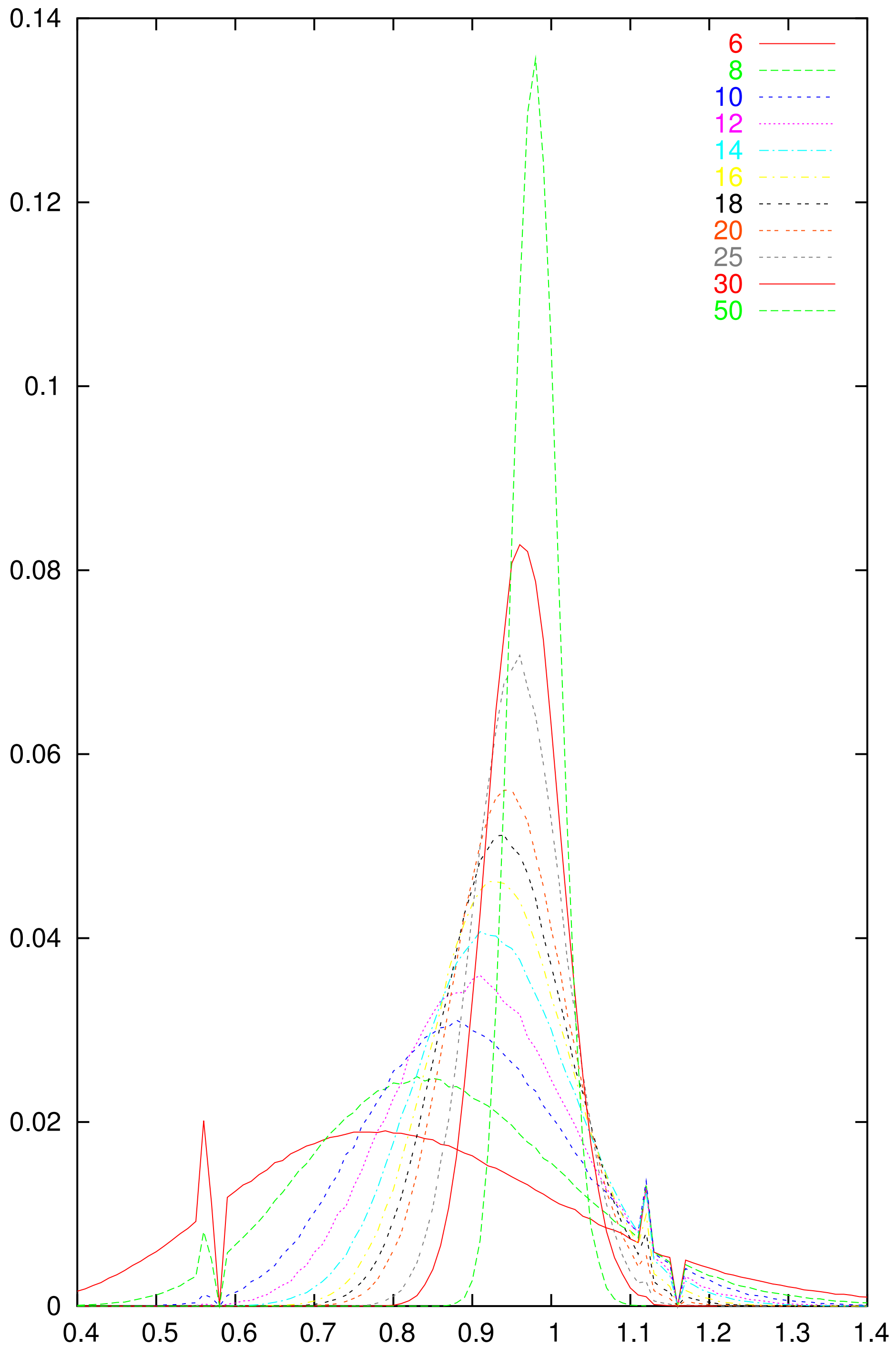

Figure 3-5: PDF of KL as a function of states for matrices generated using Russell's algorithm. The higher the number of Markov states, the more the shape resembles an impulse at 1

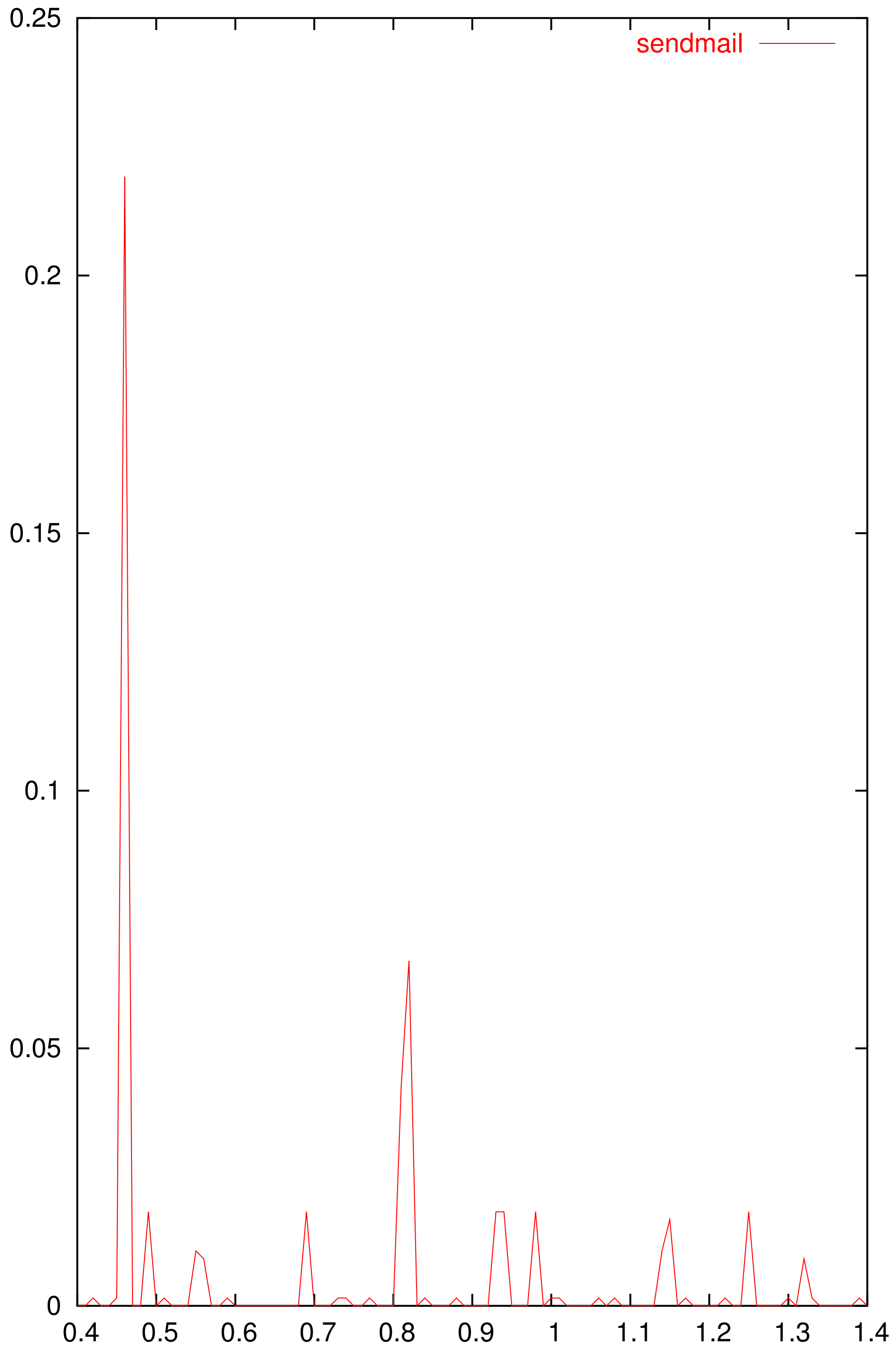

Figure 3-6: PDF of KL as a function of states for sendmail processes. Note that the density is focussed around 0.45 and that the shape is highly irregular

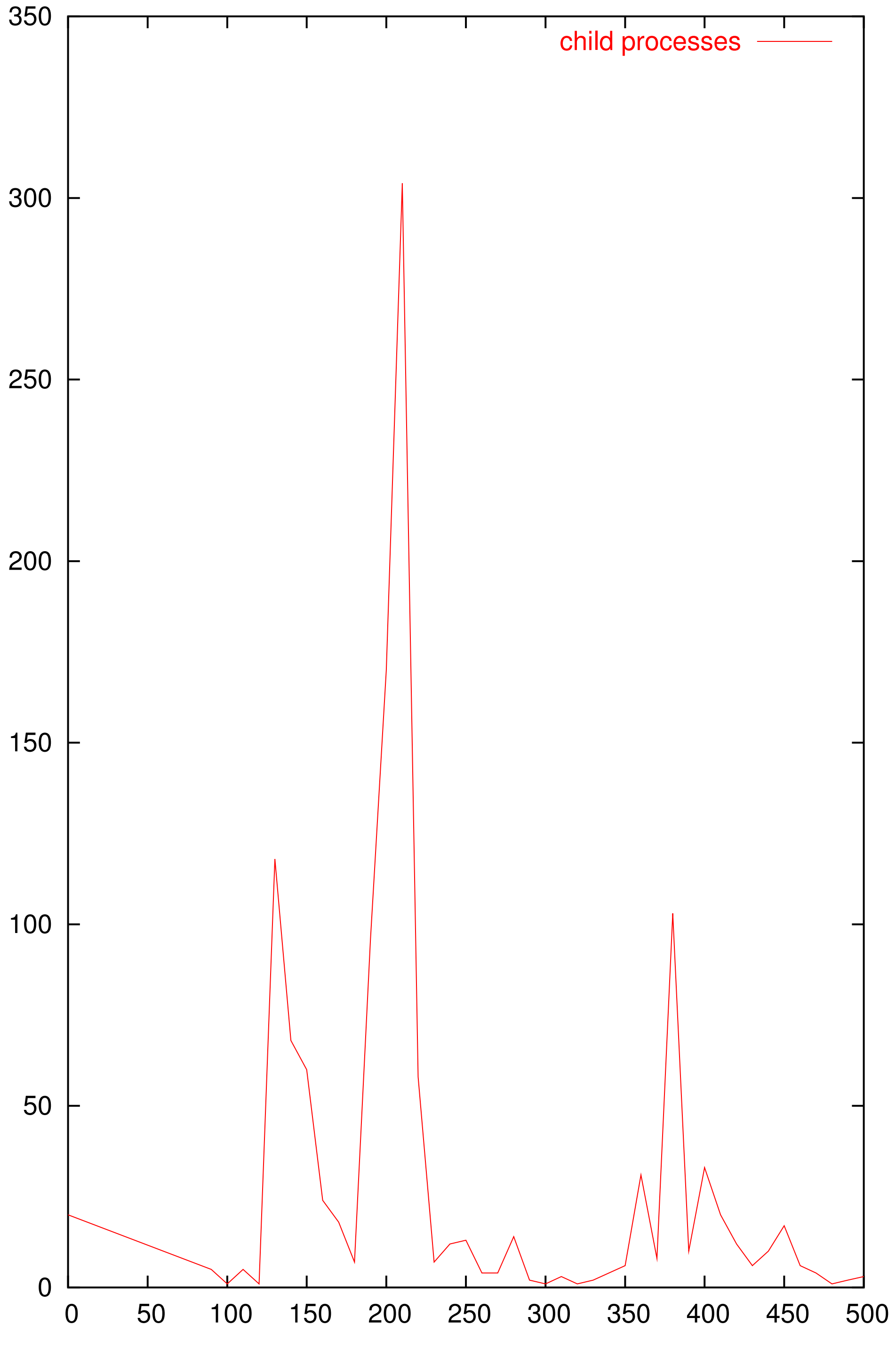

Figure 3-7: Unnormalized histogram of system calls per child process

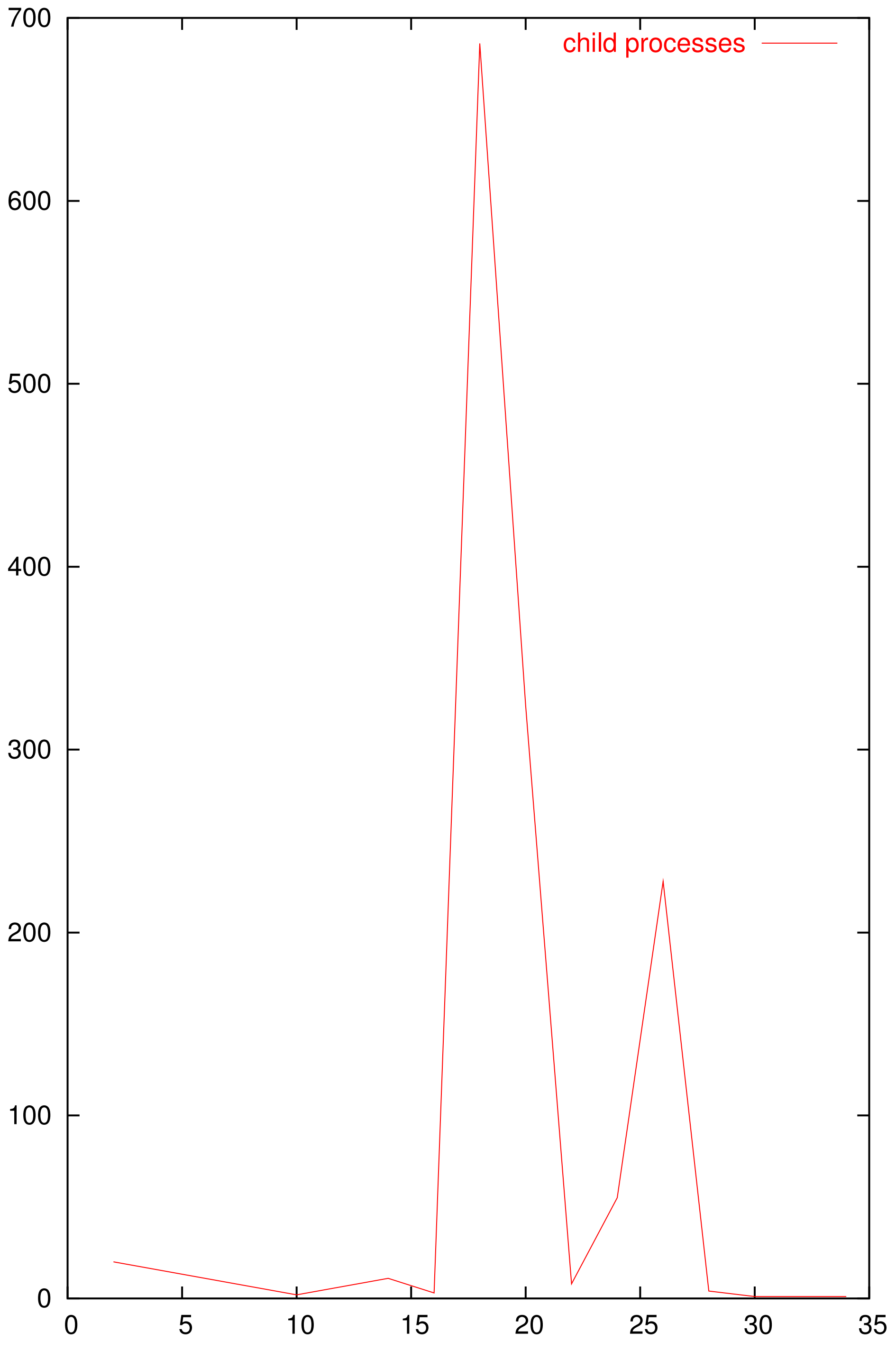

Figure 3-8: Unnormalized histogram of unique system calls per child process

3.2 Synthetic data-sets

In order to create synthetic data-sets perl programs were written using `Math::Random`, available from `http://www.cpan.org`. Russell's algorithm (1.9.2) was implemented in order to create random matrices of an arbitrary size. Figure 3-9 describes the data-sets used in this section. The sampling column describes filtering that was applied to the matrices generated. A sampling of "uniform" indicates that all matrices were accepted into the library. A sampling of $KL < 1$ indicates that matrices were chosen with the condition that the pair-wise KL scores all be less than 1. Similarly, $KL < .75$ indicates that the matrices had pair-wise KL scores of less than 0.75. These conditions were met incrementally, meaning that the first matrix was always added to the library, and each subsequent matrix was added only if the condition held with the existing matrices.

File	Matrices	States	Sampling	Execution length	Cycles	Transitions
ds1	10	6	uniform	1000	10	100,000
ds2	10	6	uniform	100	10	10,000
ds3	10	6	uniform	50	10	5,000
ds4	10	6	uniform	10	10	1,000
ds5	10	12	uniform	1000	10	100,000
ds6	10	12	uniform	100	10	10,000
ds7	10	12	uniform	50	10	5,000
ds8	10	12	uniform	10	10	1,000
ds9	10	18	uniform	1000	10	100,000
ds10	10	18	uniform	100	10	10,000
ds11	10	18	uniform	50	10	5,000
ds12	10	18	uniform	10	10	1,000
ds13	10	6	$KL < 1$	1000	10	100,000
ds14	10	6	$KL < 1$	100	10	10,000
ds15	10	6	$KL < 1$	50	10	5,000
ds16	10	6	$KL < 1$	10	10	1,000
ds17	10	12	$KL < 1$	1000	10	100,000
ds18	10	12	$KL < 1$	100	10	10,000
ds19	10	12	$KL < 1$	50	10	5,000
ds20	10	12	$KL < 1$	10	10	1,000
ds21	10	18	$KL < 1$	1000	10	100,000
ds22	10	18	$KL < 1$	100	10	10,000
ds23	10	18	$KL < 1$	50	10	5,000
ds24	10	18	$KL < 1$	10	10	1,000
ds25	10	6	$KL < .75$	1000	10	100,000
ds26	10	6	$KL < .75$	100	10	10,000
ds27	10	6	$KL < .75$	50	10	5,000
ds28	10	6	$KL < .75$	10	10	1,000
ds29	10	12	$KL < .75$	1000	10	100,000
ds30	10	12	$KL < .75$	100	10	10,000
ds31	10	12	$KL < .75$	50	10	5,000
ds32	10	12	$KL < .75$	10	10	1,000
ds33	10	18	$KL < .75$	1000	10	100,000
ds34	10	18	$KL < .75$	100	10	10,000
ds35	10	18	$KL < .75$	50	10	5,000
ds36	10	18	$KL < .75$	10	10	1,000

Figure 3-9: Synthetic data-sets

3.3 Determining β

The first parameter that was tuned was the cutoff threshold, β, used in break-point detection. This parameter specifies how many standard deviations are required between the $\texttt{score}_t$ and $\texttt{mean}_t$ variables as discussed in (1.7.3). Its optimal value is viewed as a function of the size of the matrices (s), the length that the oracle allows each matrix to execute (k), and the level of similarity between the matrices, as measured by pair-wise KL distance. In principle, if the set M of matrices were known and k could be parameterized, break-point detection could be phrased in terms of a simple frequentist hypothesis test. Given a set of Markov matrices, a sequence of transitions, and a cost function associated with changing between matrices,[1] an optimal value of β could be determined based upon the desired specificity and accuracy. However, given that these assumptions do not hold in the real-world, β was set based upon the following experiments.

3.3.1 Optimal β based on segment length

Setup

In the first experiment, $s \times s$ random matrices were generated and executed for k transitions. The output was evaluated solely on the basis of the segmentation phase of IBS. β was allowed to range over a wide range of values for each data-set. The average segment length generated by IBS was compared to k for various values of s, k, and β. Ideally, if segmentation were perfect (and non-causal in the sense that it anticipated decisions made by the oracle), the average segment length would match k exactly. The value of β used by the original authors was 2.58 (which was optimized for 5×5 matrices). This value, along with the integers between 1 and 7, was tested for each data-set with k ranging from 10 to 1000. In this and all subsequent experiments s was 6, 12, or 18.

[1]A penalty for switching matrices is necessary because the optimal solution would otherwise choose the matrix at each step with the highest value for the particular transition involved

Results

Section A-1 of the appendix show the optimal value of β within each sub-experiment. The optimal choice for β appears to depend much more upon k than upon s. Unfortunately, k is not known to the algorithm, making the choice for β much more complicated. However, several conclusions can be made from these results. First, it appears that a value between 2 and 3 is optimal for reasonable values of s and k. Second, the biased matrices (results are shown only for $s = 18$ but were similar for 6 and 12) did not behave significantly differently from the uniformly sampled matrices. This means that the segmentation phase of IBS was relatively unaffected by the bias introduced in certain data-sets.

3.3.2 Clustering accuracy

Setup

This set of experiments evaluated each of the data-sets used in the preceding section by measuring overall clustering accuracy. Accuracy was defined as the number of transitions that were correctly labeled. Each data-set consisted of 10 matrices, meaning that the correct label for each transition was an element of $[1 - 10]$. The label was relative to the the position of the cluster within the library of matrices. In other words the first segment and all subsequent subsumptions into that segment were labeled as a 1, while the second novel cluster was labeled as a 2. This form of measuring accuracy is particularly strict because it relies upon the algorithm discovering the 10 distinct clusters, which requires both effective segmentation and effective clustering. Each sub-experiment contains an entry labeled "optimal" which tested only the clustering portion of the algorithm. These tests were implemented by using data-set annotations rather than IBS segmentation. IBS was run with three forms of clustering: the standard Bayesian approach as described in the derivation, an MDL algorithm with s^2 degrees of freedom called MDL1, and an MDL algorithm with s degrees of freedom called MDL2.

Results

Section A-2 of the appendix shows the results of this set of experiments. The first conclusion drawn from this experiment is that the MDL approach to clustering is effective under a much smaller range of conditions. It had been hoped that MDL would provide a second means of running IBS in order to operate two monitors that were fairly independent of one-another. However, the poor results of MDL on these data-sets suggests that the Bayesian approach should be used at all times. The MDL approach has been extremely effective in analyzing sequences of DNA in which the number of states for each transition is 4. The fact that this clustering approach is optimized for 4 states may explain why it performed relatively well on the data-sets with 6 states while failing to perform any better than random guessing when s was 12 or 18.

Ignoring the MDL results for the remainder of this thesis, many other conclusions can be made. First, the results show that IBS is highly ineffective when $k = 10$, as expected. At this value of k even the optimal segmentation fails to produce good clusterings. This terrible performance is due to the fact that 10 transitions is not enough information to distinguish between two stochastic processes that are characterized by many independent parameters. Similarly, the results are exceptionally promising when $k = 1000$. For an appropriate value of β, IBS was able to achieve over 80% accuracy each time that unbiased matrices were used. However, the most relevant values of k are 50 and 100, which are meant to simulate conditions that would be present in sendmail traces.

Results for $k = 50$ and $k = 100$ are broken into two types: those with unbiased matrices and those with biased matrices. The unbiased case is discussed first. Figures A-9, A-12, and A-15 show the results when IBS is applied to uniformly sampled matrices. Unfortunately, despite performing extremely well with optimal clustering, IBS achieves only 10-20% accuracy for these values of k. These results indicate that the segmentation portion of the algorithm is failing to find the appropriate break-points. Referring back to the previous experiment, which showed that the average segment length in these conditions was roughly correct, this indicates that the lag between true break-points and their detection introduces enough noise to fool the clustering portion of the algorithm. This result is excellent motivation for the next experiment, in which pre-loaded matrices are used to prevent such noise from destroying clustering accuracy.

A comparison of results for biased matrices to the uniformly sampled matrices shows that the bias does indeed hinder IBS performance. Although the previous experiment showed that the biases do not systematically increase average segment length, they are detrimental to the clustering phase of the algorithm. The topic of optimal clustering when the stochastic processes are known to be biased is a very active area of mathematical research, and will likely produce better algorithms at some point in the future. However, for the purposes of this thesis, these results are taken as further incentive to use pre-loaded matrices in order to increase clustering performance.

3.4 Pre-loaded Libraries

As discussed previously, the concept behind library pre-loading is to prime the set of clusters with suitable matrices such that the initial subsumptions that IBS makes do not combine segments that should be placed into different clusters. When the library is small and segments are short, all of the generated matrices are dominated by priors and therefore look very similar in KL space. However, introducing matrices into the library helps segments to be added to the appropriate cluster rather than combined with unrelated segments to produce clusters that represent a hybrid of separate stochastic processes.

Matrix pre-loading was accomplished through data-set annotations that disabled IBS segmentation during the pre-loading phase. In other words, a data-set was constructed that contained a series of transitions generated by each underlying process. These data-sets were annotated such that IBS would automatically assign each one to its own cluster. The strength of the pre-loading could be adjusted by varying the length of each such sequence. After the pre-load sequences, the data-set was annotated to instruct IBS to begin normal segmentation, with the effect being as if the earlier transitions had been hard-coded into its internal matrices.

Setup

In this experiment 10 matrices were generated using the same types of biases as the previous experiments. For each library, data sets were constructed in which each matrix had a pre-load execution length of 0, 10, 100, 1000, or 10000. After the pre-load transitions, each matrix was again executed for 0, 10, 100, 1000, or 10000 transitions for a total of 10 cycles. The experiment was repeated for $s = 6, 12, 18$, for uniform sampling and both types of bias, and for values of 2, 2.58, 3, 4 and "optimal" for β.

Results

The results of this experiment are shown in figures A-21 - A-28. Unfortunately, results are not available for $s = 18$ and $KL < 0.75$ because this experiment was terminated after 2 weeks of execution time. Overall this experiment consumed over 2 months worth of computer time, split across 16 processors (8 computers) each running for significant portions of a 2 week period. The results of this experiment suggest that an appropriate level of pre-loading can be used to increase the performance of IBS. However, they also show that incorrect usage of pre-loading can actually diminish the clustering accuracy in the long-run, which was an unexpected result. The discussion is divided into the case in which $s = 6$ versus $s = 12$ and $s = 18$.

In the case where $s = 6$, shown in figures A-21 through A-23, a pattern emerges that adding pre-loading sequences of length 100 or 1000 increases performance while adding 10 or 10000 hurts performance. Initially, these results were believed to be flawed, but careful analysis suggests a reason for this pattern. When no pre-loading is used, the first segment is always labeled “1”, and similarly, subsequent clusters are numbered in sequence. However, when a very small amount of pre-loading is used, the pre-load matrices occupy the first 10 positions. Should the first segment of real data not be subsumed into any of the pre-load matrices, it will be labeled as “11” and forever lower accuracy even when the algorithm correctly attributes later segments generated from the same stochastic process to that segment. This phenomenon explains why having very short pre-load sequences can hurt performance as an artifact of the manner in which performance is measured. Similarly, adding pre-load sequences that are too long produces a type of over-fitting in which short segments produced later in the data-set may not be added to the correct cluster due to sampling variations. For example, if the pre-load length is 10000 but the execution length is only 10 or 100, it is likely that some of the segments will be incorrectly added as new clusters simply because they happen to diverge from the strongly entrenched cluster. Again, they are added with values greater than 10 and will forever hurt accuracy when subsequent segments are subsumed into them. Comparing uniformly sampled matrices to biased matrices for the case in which $s = 6$ shows that the effects of the bias are strongest when pre-loading is not used. This is a very fortunate result as the concept of pre-loading was introduced to combat the problem that sendmail processes tend to be highly biased.

When $s = 12$ or $s = 18$, similar patterns emerge except that the optimal threshold for pre-loading move up from 100 or 1000 to 1000 or 10000. This phenomenon makes sense as larger matrices require much more data in order to become stable. A 6x6 matrix has 30 parameters while a 12x12 has 132 and an 18x18 has 306. Given these relationships, and the arguments in the previous paragraph, it is reasonable that larger pre-loading strengths should be used when the number of states is higher. Unfortunately, not enough computer time was available to fully explore the space of parameters in order to fully optimize the process for use in an IDS.

Chapter 4

Conclusion

I began this thesis with the intention of constructing an IDS using the MAITA framework and data-monitors built around the IBS algorithm. However, the entire task of constructing an IDS according to the priniciples outlined in the first chapter was too complex of an undertaking for an MEng thesis. Therefore, I decided to focus on the analysis of IBS and its performance on synthetic data-sets under a wide range of inputs and parameterizations. This chapter summarizes the chief contributions of this thesis.

The first major contribution of this thesis is the porting of the IBS algorithm from common-LISP to java, C++, and C. In addition, the parsing engine for the code was substantially re-written in order to support many types of data-set annotation. Such annotations were used, for instance, to distinguish between training and classification data, to allow for forced-break points in the segmentation portion of the algorithm, and to inject pre-loaded matrices into the algorithm's library. In addition, the C version of the code was written to support parallelized operations on an MPI-compatible super-computer such as beowulf.lcs.mit.edu. Such parallelization was used to reduce training time and made several of the experiments described in the previous chapter much more feasible.

Several algorithmic options were added to the implementation of IBS. First, the concept of forced-break points was added in order to take advantage of the fact that system call traces also contain a time-series of PIDs. The points at which PID changes represent known discontinuities in the time-series and IBS was made aware of these events through data-set annotations. In addition, the option to perform MDL clustering rather than the standard Bayesian clustering was added. This option makes sense only for very small values of s, which is not particularly relevant for use in an IDS, but could be very useful if the code were applied to genomic data-sets.

A large corpus of code was written for this thesis that related to pre-processing of system-call traces. Initial analysis indicated that a great deal of state reduction would need to be performed in order to make use of the IBS algorithm on system-call data. Code was written in order to parse SunOS truss files, applying a number of rules discussed in the implementation chapter, in order to produce files suitable for input to IBS. Finally, a great deal of code was written in order to develop synthetic data-sets such as those used in the experiments section. The construction of "random" matrices turned out to be a much more subtle problem than originally believed, as discussed in section (1.9.2).

Unfortunately, the largest obstacle between the current implementation and a complete IDS is the need for automated data-pre-processing. This thesis has demonstrated that the number of Markov states used to characterize process behavior must be reduced significantly below the number of system calls that a process actually makes. A great deal of state reduction was performed manually for sendmail, as described in the experiments section. However these steps would need to be made more rigorous and mathematically justified before expanding the code to analyze other unix processes. In addition, tools would need to be developed for scanning executable programs for the set of system calls that they make.[1]

One possible avenue for state reduction that is particularly appealing is the use of grammar-learning algorithms such as sequitur [19]. Such algorithms attempt to describe a time-series by learning common types of components that are used to construct the series. The assumption is that the series contains sub-structures that are combined in a systematic manner. For example, a sendmail trace contains many

[1] Although this scanning is not possible in the most general sense, it should be possible to establish a superset of the calls that a program will make, assuming that it does not use self-modifying code

sections that correspond to reading a file from a socket and then writing that file to disk. These sections would appear in the trace as a long series of `read` calls followed by a long series of `write` calls. Unfortunately, this higher level pattern is not evident to IBS, which only looks at the transition probabilities between consecutive pairs of calls when trying to characterize the process. A grammar learning approach could be used to form a better set of pseudo-states than those used by the current implementation of IBS.

Appendix A

Experimental Results

A.1 Average segment length vs. β

The following experiments show the relationship between β and the average segment length produced by the segmentation portion of the IBS algorithm. The value k represents the actual value of the segment lengths used to generate the data-sets. It ranges from 10 to 1000. Ideally, one would expect the average segment length to match k. The optimal value of β is defined to be the choice that produces an average segment length that is closest to k.

File	States	β	Average Segment Length	k
ds1	6	1.0000	7.0781	1000
ds1	6	2.0000	302.1118	1000
ds1	6	2.5800	523.5550	1000
ds1	6	3.0000	704.2183	1000
ds1	**6**	**4.0000**	**1162.7791**	**1000**
ds1	6	5.0000	1515.1364	1000
ds1	6	6.0000	1886.7736	1000
ds1	6	7.0000	2272.7045	1000
ds2	6	1.0000	5.9624	100
ds2	**6**	**2.0000**	**90.0811**	**100**
ds2	6	2.5800	133.3200	100
ds2	6	3.0000	192.2885	100
ds2	6	4.0000	285.6857	100
ds2	6	5.0000	384.5769	100
ds2	6	6.0000	499.9500	100
ds2	6	7.0000	769.1538	100
ds3	6	1.0000	5.8060	50
ds3	**6**	**2.0000**	**69.4306**	**50**
ds3	6	2.5800	108.6739	50
ds3	6	3.0000	131.5526	50
ds3	6	4.0000	192.2692	50
ds3	6	5.0000	238.0476	50
ds3	6	6.0000	384.5385	50
ds3	6	7.0000	833.1667	50
ds4	**6**	**1.0000**	**4.8495**	**10**
ds4	6	2.0000	37.0000	10
ds4	6	2.5800	49.9500	10
ds4	6	3.0000	66.6000	10
ds4	6	4.0000	99.9000	10
ds4	6	5.0000	142.7143	10
ds4	6	6.0000	333.0000	10
ds4	6	7.0000	999.0000	10

Figure A-1: Average segment length vs. β for matrices with 6 states

File	States	β	Average Segment Length	k
ds5	12	1.0000	6.2499	1000
ds5	12	2.0000	383.1379	1000
ds5	12	2.5800	813.0000	1000
ds5	12	3.0000	917.4220	1000
ds5	**12**	**4.0000**	**999.9900**	**1000**
ds5	12	5.0000	1030.9175	1000
ds5	12	6.0000	1282.0385	1000
ds5	12	7.0000	1612.8871	1000
ds6	12	1.0000	5.7499	100
ds6	**12**	**2.0000**	**88.4867**	**100**
ds6	12	2.5800	156.2344	100
ds6	12	3.0000	172.3966	100
ds6	12	4.0000	256.3846	100
ds6	12	5.0000	312.4688	100
ds6	12	6.0000	370.3333	100
ds6	12	7.0000	476.1429	100
ds7	12	1.0000	6.2960	50
ds7	**12**	**2.0000**	**64.0897**	**50**
ds7	12	2.5800	104.1458	50
ds7	12	3.0000	124.9750	50
ds7	12	4.0000	178.5357	50
ds7	12	5.0000	227.2273	50
ds7	12	6.0000	294.0588	50
ds7	12	7.0000	333.2667	50
ds8	**12**	**1.0000**	**4.5409**	**10**
ds8	12	2.0000	43.4348	10
ds8	12	2.5800	71.3571	10
ds8	12	3.0000	83.2500	10
ds8	12	4.0000	142.7143	10
ds8	12	5.0000	166.5000	10
ds8	12	6.0000	249.7500	10
ds8	12	7.0000	249.7500	10

Figure A-2: Average segment length vs. β for matrices with 12 states

File	States	β	Average Segment Length	k
ds9	18	1.0000	26.0958	1000
ds9	18	2.0000	515.4588	1000
ds9	18	2.5800	781.2422	1000
ds9	18	3.0000	943.3868	1000
ds9	**18**	**4.0000**	**999.9900**	**1000**
ds9	18	5.0000	1011.9900	1000
ds9	18	6.0000	1098.8901	1000
ds9	18	7.0000	1388.8750	1000
ds10	18	1.0000	22.3691	100
ds10	**18**	**2.0000**	**109.8791**	**100**
ds10	18	2.5800	163.9180	100
ds10	18	3.0000	212.7447	100
ds10	18	4.0000	270.2432	100
ds10	18	5.0000	344.7931	100
ds10	18	6.0000	416.6250	100
ds10	18	7.0000	476.1429	100
ds11	**18**	**1.0000**	**19.8373**	**50**
ds11	18	2.0000	81.9508	50
ds11	18	2.5800	104.1458	50
ds11	18	3.0000	151.4848	50
ds11	18	4.0000	208.2917	50
ds11	18	5.0000	263.1053	50
ds11	18	6.0000	333.2667	50
ds11	18	7.0000	384.5385	50
ds12	**18**	**1.0000**	**18.8491**	**10**
ds12	18	2.0000	62.4375	10
ds12	18	2.5800	99.9000	10
ds12	18	3.0000	142.7143	10
ds12	18	4.0000	166.5000	10
ds12	18	5.0000	249.7500	10
ds12	18	6.0000	333.0000	10
ds12	18	7.0000	333.0000	10

Figure A-3: Average segment length vs. β for matrices with 18 states

File	sampling	States	β	Average Segment Length	k
ds21	$KL < 1$	18	1.0000	24.8321	1000
ds21	$KL < 1$	18	2.0000	440.5242	1000
ds21	$KL < 1$	18	2.5800	740.7333	1000
ds21	$KL < 1$	18	3.0000	943.3868	1000
ds21	$KL < 1$	**18**	**4.0000**	**990.0891**	**1000**
ds21	$KL < 1$	18	5.0000	1020.3980	1000
ds21	$KL < 1$	18	6.0000	1149.4138	1000
ds21	$KL < 1$	18	7.0000	1428.5571	1000
ds22	$KL < 1$	18	1.0000	23.1458	100
ds22	$KL < 1$	**18**	**2.0000**	**112.3483**	**100**
ds22	$KL < 1$	18	2.5800	153.8308	100
ds22	$KL < 1$	18	3.0000	204.0612	100
ds22	$KL < 1$	18	4.0000	285.6857	100
ds22	$KL < 1$	18	5.0000	357.1071	100
ds22	$KL < 1$	18	6.0000	416.6250	100
ds22	$KL < 1$	18	7.0000	499.9500	100
ds23	$KL < 1$	**18**	**1.0000**	**22.1195**	**50**
ds23	$KL < 1$	18	2.0000	94.3208	50
ds23	$KL < 1$	18	2.5800	124.9750	50
ds23	$KL < 1$	18	3.0000	151.4848	50
ds23	$KL < 1$	18	4.0000	208.2917	50
ds23	$KL < 1$	18	5.0000	277.7222	50
ds23	$KL < 1$	18	6.0000	333.2667	50
ds23	$KL < 1$	18	7.0000	384.5385	50
ds24	$KL < 1$	**18**	**1.0000**	**20.8125**	**10**
ds24	$KL < 1$	18	2.0000	58.7647	10
ds24	$KL < 1$	18	2.5800	76.8462	10
ds24	$KL < 1$	18	3.0000	90.8182	10
ds24	$KL < 1$	18	4.0000	142.7143	10
ds24	$KL < 1$	18	5.0000	199.8000	10
ds24	$KL < 1$	18	6.0000	199.8000	10
ds24	$KL < 1$	18	7.0000	333.0000	10

Figure A-4: Average segment length vs. β for biased ($KL < 1$) matrices with 18 states

File	Sampling	States	β	Average Segment Length	k
ds33	$KL < 0.75$	18	1.0000	26.9612	1000
ds33	$KL < 0.75$	18	2.0000	298.5045	1000
ds33	$KL < 0.75$	18	2.5800	529.0952	1000
ds33	$KL < 0.75$	18	3.0000	757.5682	1000
ds33	$KL < 0.75$	18	4.0000	970.8641	1000
ds33	$KL < 0.75$	**18**	**5.0000**	**999.9900**	**1000**
ds33	$KL < 0.75$	18	6.0000	1125.9900	1000
ds33	$KL < 0.75$	18	7.0000	1249.9875	1000
ds34	$KL < 0.75$	18	1.0000	22.3691	100
ds34	$KL < 0.75$	**18**	**2.0000**	**106.3723**	**100**
ds34	$KL < 0.75$	18	2.5800	151.5000	100
ds34	$KL < 0.75$	18	3.0000	192.2885	100
ds34	$KL < 0.75$	18	4.0000	270.2432	100
ds34	$KL < 0.75$	18	5.0000	333.3000	100
ds34	$KL < 0.75$	18	6.0000	399.9600	100
ds34	$KL < 0.75$	18	7.0000	476.1429	100
ds35	$KL < 0.75$	**18**	**1.0000**	**22.0220**	**50**
ds35	$KL < 0.75$	18	2.0000	86.1897	50
ds35	$KL < 0.75$	18	2.5800	116.2558	50
ds35	$KL < 0.75$	18	3.0000	147.0294	50
ds35	$KL < 0.75$	18	3.0000	147.0294	50
ds35	$KL < 0.75$	18	4.0000	227.2273	50
ds35	$KL < 0.75$	18	5.0000	277.7222	50
ds35	$KL < 0.75$	18	6.0000	357.0714	50
ds35	$KL < 0.75$	18	7.0000	454.4545	50
ds36	$KL < 0.75$	**18**	**1.0000**	**21.7174**	**10**
ds36	$KL < 0.75$	18	2.0000	62.4375	10
ds36	$KL < 0.75$	18	2.5800	90.8182	10
ds36	$KL < 0.75$	18	3.0000	142.7143	10
ds36	$KL < 0.75$	18	4.0000	199.8000	10
ds36	$KL < 0.75$	18	5.0000	249.7500	10
ds36	$KL < 0.75$	18	6.0000	333.0000	10
ds36	$KL < 0.75$	18	7.0000	333.0000	10

Figure A-5: Average segment length vs. β for biased ($KL < 0.75$) matrices with 18 states

A.1.1 Summary

File	Sampling	States	β	Average Segment Length	k
ds1	uniform	6	4.0000	1162.7791	1000
ds2	uniform	6	2.0000	90.0811	100
ds3	uniform	6	2.0000	69.4306	50
ds4	uniform	6	1.0000	4.8495	10
ds13	$KL < 1$	6	3.0000	952.3714	1000
ds14	$KL < 1$	6	2.0000	87.7105	100
ds15	$KL < 1$	6	2.0000	59.5119	50
ds16	$KL < 1$	6	1.0000	3.9801	10
ds25	$KL < 0.75$	6	4.0000	1282.0385	1000
ds26	$KL < 0.75$	6	2.0000	84.0252	100
ds27	$KL < 0.75$	6	2.0000	51.5361	10
ds28	$KL < 0.75$	6	1.0000	4.6037	1000

Figure A-6: Optimal choice of β for matrices with 6 states

File	Sampling	States	β	Average Segment Length	k
ds5	uniform	12	4.0000	999.9900	1000
ds6	uniform	12	2.0000	88.4867	100
ds7	uniform	12	2.0000	64.0897	50
ds8	uniform	12	1.0000	4.5409	10
ds17	$KL < 1$	12	4.0000	990.0891	1000
ds18	$KL < 1$	12	2.0000	78.1172	100
ds19	$KL < 1$	12	2.0000	69.4306	50
ds20	$KL < 1$	12	1.0000	5.9112	10
ds29	$KL < 0.75$	12	4.0000	1010.0909	1000
ds30	$KL < 0.75$	12	2.0000	91.7339	100
ds31	$KL < 0.75$	12	2.0000	78.1094	50
ds32	$KL < 0.75$	12	1.0000	4.0445	10

Figure A-7: Optimal choice of β for matrices with 12 states

File	Sampling	States	β	Average Segment Length	k
ds9	uniform	18	4.0000	999.9900	1000
ds10	uniform	18	2.0000	109.8791	100
ds11	uniform	18	1.0000	19.8373	50
ds12	uniform	18	1.0000	18.8491	10
ds21	$KL < 1$	18	4.0000	990.0891	1000
ds22	$KL < 1$	18	2.0000	112.3483	100
ds23	$KL < 1$	18	1.0000	22.1195	50
ds24	$KL < 1$	18	1.0000	20.8125	10
ds33	$KL < 0.75$	18	5.0000	999.9900	1000
ds34	$KL < 0.75$	18	2.0000	106.3723	100
ds35	$KL < 0.75$	18	1.0000	22.0220	50
ds36	$KL < 0.75$	18	1.0000	21.7174	10

Figure A-8: Optimal choice of β for matrices with 18 states

A.2 Clustering Performance

The following charts explore the relationship between β and the clustering performance of IBS. The accuracy of clustering is defined as the total number of transitions that are correctly labeled. MDL1 refers to MDL clustering with s^2 degrees of freedom while MDL2 refers to MDL clustering with s degrees of freedom. There are 10 matrices in each experiment, meaning that random guessing should produce an accuracy of 10%. The optimal row in each of the charts refers to optimal segmentation based upon break-point annotation in the data-sets. In other words, these particular executions of IBS do not use the segmentation portion of the algorithm and therefore provide a baseline accuracy based solely on the clustering portion of the algorithm. After performing the experiment on the data-sets with 6 states, the number of β values was reduced based upon the observation that values below 2 or greater than 4 were clearly sub-optimal.

File	Sampling	States	k	β	Bayesian	MDL1	MDL2
ds1	uniform	6	1000	1.00	0.0135	0.0747	0.0835
ds1	uniform	6	1000	2.00	0.0918	0.0918	0.2145
ds1	uniform	6	1000	2.58	0.2396	0.1821	0.3411
ds1	uniform	6	1000	3.00	**0.8149**	**0.7397**	0.3332
ds1	uniform	6	1000	4.00	0.1279	0.1279	**0.3486**
ds1	uniform	6	1000	5.00	0.0697	0.0697	0.0957
ds1	uniform	6	1000	6.00	0.0285	0.0685	0.1248
ds1	uniform	6	1000	7.00	0.0428	0.0428	0.0979
ds1	uniform	6	1000	optimal	**0.9990**	**0.9990**	**0.0999**
ds2	uniform	6	100	1.00	0.0172	0.0892	0.1053
ds2	uniform	6	100	2.00	0.1311	0.0999	0.1000
ds2	uniform	6	100	2.58	**0.2138**	0.0999	0.1000
ds2	uniform	6	100	3.00	0.2088	0.0999	0.1000
ds2	uniform	6	100	4.00	0.0621	0.0999	0.1000
ds2	uniform	6	100	5.00	0.0554	0.0999	0.1000
ds2	uniform	6	100	6.00	0.0999	0.0999	0.1000
ds2	uniform	6	100	7.00	0.0999	0.0999	0.1000
ds2	uniform	6	100	optimal	**0.8119**	**0.1989**	**0.1990**
ds3	uniform	6	50	1.00	0.0770	0.0996	0.0956
ds3	uniform	6	50	2.00	**0.2178**	0.0998	0.1000
ds3	uniform	6	50	2.58	0.0540	0.0998	0.1000
ds3	uniform	6	50	3.00	0.0752	0.0998	0.1000
ds3	uniform	6	50	4.00	0.0676	0.0998	0.1000
ds3	uniform	6	50	5.00	0.0998	0.0998	0.1000
ds3	uniform	6	50	6.00	0.0998	0.0998	0.1000
ds3	uniform	6	50	7.00	0.0998	0.0998	0.1000
ds3	uniform	6	50	optimal	**0.8724**	**0.0998**	**0.1000**
ds4	uniform	6	10	1.00	0.0591	0.0991	0.0991
ds4	uniform	6	10	2.00	0.1011	0.0991	0.1001
ds4	uniform	6	10	2.58	0.1061	0.0991	0.1001
ds4	uniform	6	10	3.00	0.0991	0.0991	0.1001
ds4	uniform	6	10	4.00	0.0991	0.0991	0.1001
ds4	uniform	6	10	5.00	0.0991	0.0991	0.1001
ds4	uniform	6	10	6.00	0.0991	0.0991	0.1001
ds4	uniform	6	10	7.00	0.0991	0.0991	0.1001
ds4	uniform	6	10	optimal	**0.3524**	**0.0991**	**0.1001**

Figure A-9: Clustering performance for uniformly sampled matrices with 6 states

File	Sampling	States	k	β	Bayesian	MDL1	MDL2
ds13	$KL < 1$	6	1000	1.00	0.0191	0.0828	0.0146
ds13	$KL < 1$	6	1000	2.00	0.1190	0.1196	0.0571
ds13	$KL < 1$	6	1000	2.58	**0.9042**	**0.9089**	0.0843
ds13	$KL < 1$	6	1000	3.00	0.8840	0.8840	0.1070
ds13	$KL < 1$	6	1000	4.00	0.7845	0.7845	0.1612
ds13	$KL < 1$	6	1000	5.00	0.4733	0.3945	0.3765
ds13	$KL < 1$	6	1000	6.00	0.0765	0.1427	0.3319
ds13	$KL < 1$	6	1000	7.00	0.0921	0.1143	0.1555
ds13	$KL < 1$	6	1000	optimal	**0.9990**	**0.9990**	**0.0009**
ds14	$KL < 1$	6	100	1.00	0.0273	0.0973	0.0995
ds14	$KL < 1$	6	100	2.00	0.1302	0.0999	0.1000
ds14	$KL < 1$	6	100	2.58	**0.1435**	0.0999	0.1000
ds14	$KL < 1$	6	100	3.00	0.1063	0.0999	0.1000
ds14	$KL < 1$	6	100	4.00	0.0863	0.0999	0.1000
ds14	$KL < 1$	6	100	5.00	0.0609	0.0999	0.1000
ds14	$KL < 1$	6	100	6.00	0.0852	0.0999	0.1000
ds14	$KL < 1$	6	100	7.00	0.0970	0.0999	0.1000
ds14	$KL < 1$	6	100	optimal	**0.7901**	**0.0999**	**0.1000**
ds15	$KL < 1$	6	50	1.00	0.0692	0.0980	0.0998
ds15	$KL < 1$	6	50	2.00	0.0432	0.0998	0.1000
ds15	$KL < 1$	6	50	2.58	**0.1408**	0.0998	0.1000
ds15	$KL < 1$	6	50	3.00	0.0994	0.0998	0.1000
ds15	$KL < 1$	6	50	4.00	0.1352	0.0998	0.1000
ds15	$KL < 1$	6	50	5.00	0.0842	0.0998	0.1000
ds15	$KL < 1$	6	50	6.00	0.0998	0.0998	0.1000
ds15	$KL < 1$	6	50	7.00	0.0998	0.0998	0.1000
ds15	$KL < 1$	6	50	optimal	**0.5979**	**0.0996**	**0.1000**
ds16	$KL < 1$	6	10	1.00	0.0460	0.0961	0.1011
ds16	$KL < 1$	6	10	2.00	**0.1221**	0.0991	0.1001
ds16	$KL < 1$	6	10	2.58	0.1091	0.0991	0.1001
ds16	$KL < 1$	6	10	3.00	0.0891	0.0991	0.1001
ds16	$KL < 1$	6	10	4.00	0.0991	0.0991	0.1001
ds16	$KL < 1$	6	10	5.00	0.1091	0.0991	0.1001
ds16	$KL < 1$	6	10	6.00	0.0991	0.0991	0.1001
ds16	$KL < 1$	6	10	7.00	0.0991	0.0991	0.1001
ds16	$KL < 1$	6	10	optimal	**0.0951**	**0.0901**	**0.0991**

Figure A-10: Clustering performance for biased matrices ($KL < 1$) with 6 states

File	Sampling	States	k	β	Bayesian	MDL1	MDL2
ds25	$KL < .75$	6	1000	2.00	0.0855	0.0999	0.3037
ds25	$KL < .75$	6	1000	2.58	0.0668	0.1011	0.0309
ds25	$KL < .75$	6	1000	3.00	**0.3897**	**0.3754**	**0.3841**
ds25	$KL < .75$	6	1000	4.00	0.2465	0.2111	0.1520
ds25	$KL < .75$	6	1000	optimal	**0.9990**	**0.8991**	**0.5997**
ds26	$KL < .75$	6	100	2.00	0.1589	0.0999	0.1000
ds26	$KL < .75$	6	100	2.58	0.1697	0.0999	0.1000
ds26	$KL < .75$	6	100	3.00	0.1289	0.0999	0.1000
ds26	$KL < .75$	6	100	4.00	0.1080	0.0999	0.1000
ds26	$KL < .75$	6	100	optimal	**0.6040**	**0.0999**	**0.1000**
ds27	$KL < .75$	6	50	2.00	0.1352	0.0998	0.1000
ds27	$KL < .75$	6	50	2.58	0.1164	0.0998	0.1000
ds27	$KL < .75$	6	50	3.00	0.0998	0.0920	0.0926
ds27	$KL < .75$	6	50	4.00	0.0998	0.0998	0.1000
ds27	$KL < .75$	6	optimal	50	**0.3829**	**0.0898**	**0.0998**
ds28	$KL < .75$	6	10	2.00	0.1011	0.0991	0.1001
ds28	$KL < .75$	6	10	2.58	0.0991	0.0991	0.1001
ds28	$KL < .75$	6	10	3.00	0.0991	0.0991	0.1001
ds28	$KL < .75$	6	10	4.00	0.0991	0.0991	0.1001
ds28	$KL < .75$	6	10	optimal	**0.1011**	**0.0991**	**0.1001**

Figure A-11: Clustering performance for biased matrices ($KL < .75$) with 6 states

File	Sampling	States	k	β	Bayesian	MDL1	MDL2
ds5	uniform	12	1000	2.00	0.0917	0.1000	0.1000
ds5	uniform	12	1000	2.58	0.5444	0.1000	0.1000
ds5	uniform	12	1000	3.00	**0.8796**	0.1000	0.1000
ds5	uniform	12	1000	4.00	0.8215	0.1000	0.1000
ds5	uniform	12	1000	optimal	**0.9990**	**0.2898**	**0.0002**
ds6	uniform	12	100	2.00	0.0805	0.0998	0.1001
ds6	uniform	12	100	2.58	0.0249	0.0999	0.1000
ds6	uniform	12	100	3.00	0.0647	0.0999	0.1000
ds6	uniform	12	100	4.00	0.0099	0.0999	0.1000
ds6	uniform	12	100	optimal	**0.9901**	**0.0900**	**0.0999**
ds7	uniform	12	50	2.00	0.0710	0.0998	0.1000
ds7	uniform	12	50	2.58	0.1020	0.0998	0.1000
ds7	uniform	12	50	3.00	0.0774	0.0998	0.1000
ds7	uniform	12	50	4.00	0.0522	0.0998	0.1000
ds7	uniform	12	50	optimal	**0.6567**	**0.1096**	**0.0902**
ds8	uniform	12	10	2.00	0.0781	0.0991	0.1001
ds8	uniform	12	10	2.58	0.1151	0.0991	0.1001
ds8	uniform	12	10	3.00	0.1081	0.0991	0.1001
ds8	uniform	12	10	4.00	0.1001	0.0991	0.1001
ds8	uniform	12	10	optimal	**0.0851**	**0.1081**	**0.1001**

Figure A-12: Clustering performance for uniformly sampled matrices with 12 states

File	Sampling	States	k	β	Bayesian	MDL1	MDL2
ds17	$KL < 1$	12	1000	2.00	0.2078	0.1000	0.1000
ds17	$KL < 1$	12	1000	2.58	0.4504	0.1000	0.1000
ds17	$KL < 1$	12	1000	3.00	0.7211	0.1000	0.1000
ds17	$KL < 1$	12	1000	4.00	**0.8483**	0.1000	0.1000
ds17	$KL < 1$	12	1000	optimal	**0.9990**	**0.1999**	**0.1999**
ds18	$KL < 1$	12	100	2.00	**0.2200**	0.0999	0.0999
ds18	$KL < 1$	12	100	2.58	0.1743	0.0999	0.1000
ds18	$KL < 1$	12	100	3.00	0.1081	0.0999	0.1000
ds18	$KL < 1$	12	100	4.00	0.1098	0.0999	0.1000
ds18	$KL < 1$	12	100	optimal	**0.9901**	**0.0999**	**0.1000**
ds19	$KL < 1$	12	50	2.00	**0.1530**	0.1004	0.1000
ds19	$KL < 1$	12	50	2.58	0.0778	0.0998	0.1000
ds19	$KL < 1$	12	50	3.00	0.1316	0.0998	0.1000
ds19	$KL < 1$	12	50	4.00	0.0724	0.0998	0.1000
ds19	$KL < 1$	12	50	optimal	**0.4999**	**0.0998**	**0.1000**
ds20	$KL < 1$	12	10	2.00	0.0681	0.0991	0.0991
ds20	$KL < 1$	12	10	2.58	0.0891	0.0991	0.1001
ds20	$KL < 1$	12	10	3.00	0.0891	0.0991	0.1001
ds20	$KL < 1$	12	10	4.00	0.1141	0.0991	0.1001
ds20	$KL < 1$	12	10	optimal	**0.1481**	**0.0981**	**0.1001**

Figure A-13: Clustering performance for biased matrices ($KL < 1$) with 12 states

File	Sampling	States	k	β	Bayesian	MDL1	MDL2
ds29	$KL < .75$	12	1000	2.00	0.1153	0.1000	0.1000
ds29	$KL < .75$	12	1000	2.58	0.1830	0.1000	0.1000
ds29	$KL < .75$	12	1000	3.00	0.1794	0.1000	0.1000
ds29	$KL < .75$	12	1000	4.00	**0.8227**	0.1000	0.1000
ds29	$KL < .75$	12	1000	optimal	**0.9990**	**0.1000**	**0.1000**
ds30	$KL < .75$	12	100	2.00	0.0713	0.0999	0.1001
ds30	$KL < .75$	12	100	2.58	0.0615	0.0999	0.1000
ds30	$KL < .75$	12	100	3.00	0.0476	0.0999	0.1000
ds30	$KL < .75$	12	100	4.00	0.1185	0.0999	0.1000
ds30	$KL < .75$	12	100	optimal	**0.5347**	**0.0999**	**0.1000**
ds31	$KL < .75$	12	50	2.00	0.1034	0.0998	0.1000
ds31	$KL < .75$	12	50	2.58	0.1626	0.0998	0.1000
ds31	$KL < .75$	12	50	3.00	0.0998	0.0998	0.1000
ds31	$KL < .75$	12	50	4.00	0.0998	0.0998	0.1000
ds31	$KL < .75$	12	50	optimal	**0.3055**	**0.0998**	**0.1000**
ds32	$KL < .75$	12	10	2.00	0.1021	0.0991	0.0991
ds32	$KL < .75$	12	10	2.58	0.1051	0.0991	0.1001
ds32	$KL < .75$	12	10	3.00	0.0951	0.0991	0.1001
ds32	$KL < .75$	12	10	4.00	0.0991	0.0991	0.1001
ds32	$KL < .75$	12	10	optimal	**0.1101**	**0.0981**	**0.0921**

Figure A-14: Clustering performance for biased matrices ($KL < 0.75$) with 12 states

File	Sampling	States	k	β	Bayesian	MDL1	MDL2
ds9	uniform	18	1000	2.00	0.1891	0.1000	0.1000
ds9	uniform	18	1000	2.58	0.3082	0.1000	0.1000
ds9	uniform	18	1000	3.00	0.2666	0.1000	0.1000
ds9	uniform	18	1000	4.00	**0.8241**	0.1000	0.1000
ds9	uniform	18	1000	optimal	**0.9990**	**0.0900**	**0.1000**
ds10	uniform	18	100	2.00	**0.2395**	0.0999	0.0999
ds10	uniform	18	100	2.58	0.2083	0.0999	0.1000
ds10	uniform	18	100	3.00	0.0778	0.0999	0.1000
ds10	uniform	18	100	4.00	0.0699	0.0999	0.1000
ds10	uniform	18	100	optimal	**0.8515**	**0.0999**	**0.1000**
ds11	uniform	18	50	2.00	0.0610	0.0998	0.1000
ds11	uniform	18	50	2.58	0.0240	0.0998	0.1002
ds11	uniform	18	50	3.00	0.0360	0.0998	0.1000
ds11	uniform	18	50	4.00	0.0850	0.0998	0.1000
ds11	uniform	18	50	optimal	**0.4313**	**0.1096**	**0.1000**
ds12	uniform	18	10	2.00	0.0791	0.0991	0.1001
ds12	uniform	18	10	2.58	0.1101	0.0991	0.1001
ds12	uniform	18	10	3.00	0.1011	0.0991	0.1001
ds12	uniform	18	10	4.00	0.1071	0.0991	0.1001
ds12	uniform	18	10	optimal	**0.1061**	**0.1081**	**0.1001**

Figure A-15: Clustering performance for uniformly sampled matrices with 18 states

File	Sampling	States	k	β	Bayesian	MDL1	MDL2
ds21	$KL < 1$	18	1000	2.00	0.0946	0.1000	0.1000
ds21	$KL < 1$	18	1000	2.58	0.3797	0.1000	0.1000
ds21	$KL < 1$	18	1000	3.00	0.4806	0.1000	0.1000
ds21	$KL < 1$	18	1000	4.00	**0.8400**	0.1000	0.1000
ds21	$KL < 1$	18	1000	optimal	**0.9890**	**0.1000**	**0.0900**
ds22	$KL < 1$	18	100	2.00	0.0300	0.0999	0.1000
ds22	$KL < 1$	18	100	2.58	0.0147	0.0999	0.1000
ds22	$KL < 1$	18	100	3.00	0.0839	0.0999	0.1000
ds22	$KL < 1$	18	100	4.00	0.0555	0.0999	0.1000
ds22	$KL < 1$	18	100	optimal	**0.8020**	**0.0999**	**0.0902**
ds23	$KL < 1$	18	50	2.00	0.0198	0.0998	0.1000
ds23	$KL < 1$	18	50	2.58	0.0532	0.0998	0.1000
ds23	$KL < 1$	18	50	3.00	0.0806	0.0998	0.1000
ds23	$KL < 1$	18	50	4.00	0.1242	0.0998	0.1086
ds23	$KL < 1$	18	50	optimal	**0.2058**	**0.1096**	**0.1000**
ds24	$KL < 1$	18	10	2.00	0.0831	0.0991	0.1001
ds24	$KL < 1$	18	10	2.58	0.1021	0.0991	0.1001
ds24	$KL < 1$	18	10	3.00	0.1191	0.0991	0.1001
ds24	$KL < 1$	18	10	4.00	0.0991	0.0991	0.1001
ds24	$KL < 1$	18	10	optimal	**0.1892**	**0.0991**	**0.0991**

Figure A-16: Clustering performance for biased ($KL < 1$) matrices with 18 states

File	Sampling	States	k	β	Bayesian	MDL1	MDL2
ds33	$KL < .75$	18	1000	2.00	0.0085	0.1000	0.1000
ds33	$KL < .75$	18	1000	2.58	0.0970	0.1000	0.1000
ds33	$KL < .75$	18	1000	3.00	0.0972	0.1000	0.1000
ds33	$KL < .75$	18	1000	4.00	0.1618	0.1000	0.1000
ds33	$KL < .75$	18	1000	optimal	**0.9990**	**0.1000**	**0.1000**
ds34	$KL < .75$	18	100	2.00	0.0726	0.0999	0.1000
ds34	$KL < .75$	18	100	2.58	0.0931	0.0999	0.1000
ds34	$KL < .75$	18	100	3.00	0.0901	0.0999	0.1000
ds34	$KL < .75$	18	100	4.00	0.0822	0.0999	0.1000
ds34	$KL < .75$	18	100	optimal	**0.7130**	**0.0999**	**0.1000**
ds35	$KL < .75$	18	50	2.00	0.0396	0.0998	0.1000
ds35	$KL < .75$	18	50	2.58	0.1032	0.0998	0.1000
ds35	$KL < .75$	18	50	3.00	0.0994	0.0998	0.1000
ds35	$KL < .75$	18	50	4.00	0.0572	0.0998	0.1000
ds35	$KL < .75$	18	50	optimal	**0.2354**	**0.1096**	**0.1000**
ds36	$KL < .75$	18	10	2.00	0.1151	0.0991	0.1001
ds36	$KL < .75$	18	10	2.58	0.0901	0.0991	0.1001
ds36	$KL < .75$	18	10	3.00	0.1021	0.0991	0.1001
ds36	$KL < .75$	18	10	4.00	0.1091	0.0991	0.1001
ds36	$KL < .75$	18	10	optimal	**0.1051**	**0.0991**	**0.1001**

Figure A-17: Clustering performance for biased ($KL < 0.75$) matrices with 18 states

File	Sampling	States	k	β	Bayesian
ds1	uniform	6	1000	3.00	0.8149
ds1	uniform	6	1000	optimal	0.9990
ds2	uniform	6	100	2.58	0.2138
ds2	uniform	6	100	optimal	0.8119
ds3	uniform	6	50	2.00	0.2178
ds3	uniform	6	50	optimal	0.8724
ds13	$KL < 1$	6	1000	2.58	0.9042
ds13	$KL < 1$	6	1000	optimal	0.9990
ds14	$KL < 1$	6	100	2.58	0.1435
ds14	$KL < 1$	6	100	optimal	0.7901
ds15	$KL < 1$	6	50	2.58	0.1408
ds15	$KL < 1$	6	50	optimal	0.5979
ds25	$KL < .75$	6	1000	3.00	0.3897
ds25	$KL < .75$	6	1000	optimal	0.9990
ds26	$KL < .75$	6	100	2.58	0.1697
ds26	$KL < .75$	6	100	optimal	0.6040
ds27	$KL < .75$	6	50	2.00	0.1352
ds27	$KL < .75$	6	50	optimal	0.3829

Figure A-18: Clustering Performance Summary for matrices with 6 states

File	Sampling	States	k	β	Bayesian
ds5	uniform	12	1000	3.00	0.8796
ds5	uniform	12	1000	optimal	0.9990
ds6	uniform	12	100	2.00	0.0805
ds6	uniform	12	100	optimal	0.9901
ds7	uniform	12	50	2.58	0.1020
ds7	uniform	12	50	optimal	0.6567
ds17	$KL < 1$	12	1000	4.00	0.8483
ds17	$KL < 1$	12	1000	optimal	0.9990
ds18	$KL < 1$	12	100	2.00	0.2200
ds18	$KL < 1$	12	100	optimal	0.9901
ds19	$KL < 1$	12	50	2.00	0.1530
ds19	$KL < 1$	12	50	optimal	0.4999
ds29	$KL < .75$	12	1000	4.00	0.8227
ds29	$KL < .75$	12	1000	optimal	0.9990
ds30	$KL < .75$	12	100	4.00	0.1185
ds30	$KL < .75$	12	100	optimal	0.5347
ds31	$KL < .75$	12	50	2.58	0.1626
ds31	$KL < .75$	12	50	optimal	0.3055

Figure A-19: Clustering Performance Summary for matrices with 12 states

File	Sampling	States	k	β	Bayesian
ds9	uniform	18	1000	4.00	0.8241
ds9	uniform	18	1000	optimal	0.9990
ds10	uniform	18	100	2.00	0.2395
ds10	uniform	18	100	optimal	0.8515
ds11	uniform	18	50	4.00	0.0850
ds11	uniform	18	50	optimal	0.4313
ds21	$KL < 1$	18	1000	4.00	0.8400
ds21	$KL < 1$	18	1000	optimal	0.9890
ds22	$KL < 1$	18	100	3.00	0.0839
ds22	$KL < 1$	18	100	optimal	0.8020
ds23	$KL < 1$	18	50	4.00	0.1242
ds23	$KL < 1$	18	50	optimal	0.2058

Figure A-20: Clustering Performance Summary for matrices with 18 states

A.3 Pre-loading Effectiveness

States	Sampling	Beta	Pre-loading	10	100	1000	10000
6	uniform	2.000000	0	0.104838	0.310035	0.182922	0.120696
6	uniform	2.000000	10	0.099200	0.125050	0.132481	0.147653
6	uniform	2.000000	100	0.046000	0.182170	0.239921	0.305219
6	uniform	2.000000	1000	0.007800	0.071530	0.171772	0.153466
6	uniform	2.000000	10000	0.000000	0.019720	0.141998	0.150960
6	uniform	2.580000	0	0.091102	0.179801	0.468938	0.333690
6	uniform	2.580000	10	0.097000	0.151870	0.172143	0.210842
6	uniform	2.580000	100	0.060000	0.332330	0.558321	0.626584
6	uniform	2.580000	1000	0.019400	0.136980	0.512998	0.491024
6	uniform	2.580000	10000	0.012000	0.068380	0.345980	0.469964
6	uniform	3.000000	0	0.090900	0.124974	0.620049	0.749664
6	uniform	3.000000	10	0.089800	0.150550	0.194669	0.214946
6	uniform	3.000000	100	0.074900	0.341810	0.670799	0.730659
6	uniform	3.000000	1000	0.056429	0.154400	0.603472	0.819076
6	uniform	3.000000	10000	0.050000	0.048340	0.511789	0.731456
6	uniform	4.000000	0	0.090900	0.103814	0.608073	0.954776
6	uniform	4.000000	10	0.090500	0.136290	0.177335	0.198318
6	uniform	4.000000	100	0.080600	0.242090	0.661241	0.833645
6	uniform	4.000000	1000	0.094600	0.063540	0.643298	0.942449
6	uniform	4.000000	10000	0.098000	0.035680	0.382132	0.939212
6	uniform	optimal	0	0.426185	0.947113	0.999100	0.999900
6	uniform	optimal	10	0.155400	0.162820	0.185795	0.197617
6	uniform	optimal	100	0.553700	0.818730	0.777671	0.723717
6	uniform	optimal	1000	0.706557	0.924660	0.833166	0.754186
6	uniform	optimal	10000	0.676327	0.932580	0.868898	0.771900

Figure A-21: Pre-loading Results for uniformly sampled matrices with 6 states

States	Sampling	Beta	Pre-loading	10	100	1000	10000
6	$KL < 1$	2.000000	0	0.094132	0.277134	0.181806	0.161876
6	$KL < 1$	2.000000	10	0.080200	0.094020	0.127324	0.103270
6	$KL < 1$	2.000000	100	0.058400	0.155820	0.226006	0.233368
6	$KL < 1$	2.000000	1000	0.006800	0.052040	0.144052	0.156996
6	$KL < 1$	2.000000	10000	0.003000	0.028180	0.146546	0.131692
6	$KL < 1$	2.580000	0	0.094536	0.180122	0.464986	0.463384
6	$KL < 1$	2.580000	10	0.094000	0.141940	0.192844	0.169758
6	$KL < 1$	2.580000	100	0.073800	0.367580	0.506030	0.558640
6	$KL < 1$	2.580000	1000	0.013600	0.136960	0.440644	0.578896
6	$KL < 1$	2.580000	10000	0.007800	0.065340	0.406996	0.384790
6	$KL < 1$	3.000000	0	0.090900	0.120438	0.688020	0.707962
6	$KL < 1$	3.000000	10	0.086400	0.149420	0.167812	0.141084
6	$KL < 1$	3.000000	100	0.079600	0.340820	0.659052	0.708692
6	$KL < 1$	3.000000	1000	0.052400	0.193960	0.702044	0.764114
6	$KL < 1$	3.000000	10000	0.052000	0.100880	0.535552	0.713450
6	$KL < 1$	4.000000	0	0.090900	0.101302	0.631052	0.939748
6	$KL < 1$	4.000000	10	0.103400	0.119460	0.158766	0.176340
6	$KL < 1$	4.000000	100	0.071000	0.247740	0.617860	0.776534
6	$KL < 1$	4.000000	1000	0.092200	0.076780	0.665332	0.923462
6	$KL < 1$	4.000000	10000	0.098000	0.064980	0.503244	0.948696
6	$KL < 1$	optimal	0	0.412710	0.929558	0.999100	0.999900
6	$KL < 1$	optimal	10	0.150000	0.172820	0.129910	0.152000
6	$KL < 1$	optimal	100	0.522000	0.807900	0.703296	0.679920
6	$KL < 1$	optimal	1000	0.687200	0.936540	0.801198	0.767902
6	$KL < 1$	optimal	10000	0.693600	0.916740	0.841158	0.773900

Figure A-22: Pre-loading Results for biased ($KL < 1$) matrices with 6 states

States	Sampling	Beta	Pre-loading	10	100	1000	10000
6	$KL < 0.75$	2.000000	0	0.098374	0.245028	0.167956	0.139786
6	$KL < 0.75$	2.000000	10	0.086600	0.111120	0.110880	0.116248
6	$KL < 0.75$	2.000000	100	0.071000	0.147700	0.205678	0.169892
6	$KL < 0.75$	2.000000	1000	0.011000	0.089120	0.179616	0.180746
6	$KL < 0.75$	2.000000	10000	0.015600	0.039480	0.133430	0.161658
6	$KL < 0.75$	2.580000	0	0.091506	0.148704	0.361226	0.316879
6	$KL < 0.75$	2.580000	10	0.088800	0.125120	0.161004	0.120348
6	$KL < 0.75$	2.580000	100	0.094800	0.351680	0.485468	0.484768
6	$KL < 0.75$	2.580000	1000	0.024200	0.119860	0.425026	0.395516
6	$KL < 0.75$	2.580000	10000	0.010000	0.063940	0.236226	0.288912
6	$KL < 0.75$	3.000000	0	0.090900	0.128490	0.619214	0.464042
6	$KL < 0.75$	3.000000	10	0.089600	0.124040	0.153430	0.159908
6	$KL < 0.75$	3.000000	100	0.074600	0.296380	0.600596	0.700000
6	$KL < 0.75$	3.000000	1000	0.028800	0.169520	0.546488	0.622898
6	$KL < 0.75$	3.000000	10000	0.022000	0.093520	0.428872	0.570220
6	$KL < 0.75$	4.000000	0	0.090900	0.099300	0.521052	0.919784
6	$KL < 0.75$	4.000000	10	0.097000	0.136040	0.164100	0.203168
6	$KL < 0.75$	4.000000	100	0.082800	0.257960	0.595680	0.667280
6	$KL < 0.75$	4.000000	1000	0.091400	0.112660	0.660410	0.893910
6	$KL < 0.75$	4.000000	10000	0.090000	0.073840	0.536732	0.903080
6	$KL < 0.75$	optimal	0	0.410890	0.858206	0.999100	0.999900
6	$KL < 0.75$	optimal	10	0.119200	0.170840	0.139886	0.158000
6	$KL < 0.75$	optimal	100	0.433000	0.722720	0.607396	0.621930
6	$KL < 0.75$	optimal	1000	0.617800	0.914760	0.773226	0.715904
6	$KL < 0.75$	optimal	10000	0.614000	0.940500	0.831168	0.737900

Figure A-23: Pre-loading results for biased ($KL < 0.75$) matrices with 6 states

States	Sampling	Beta	Pre-loading	10	100	1000	10000
12	uniform	2.000000	0	0.115847	0.190524	0.127807	0.123910
12	uniform	2.000000	10	0.057400	0.065010	0.062226	0.058178
12	uniform	2.000000	100	0.025300	0.078947	0.133860	0.131530
12	uniform	2.000000	1000	0.007600	0.039460	0.110261	0.131960
12	uniform	2.000000	10000	0.006000	0.016880	0.098664	0.112774
12	uniform	2.580000	0	0.104232	0.232582	0.195337	0.188170
12	uniform	2.580000	10	0.073100	0.087250	0.086736	0.102572
12	uniform	2.580000	100	0.037600	0.169600	0.296004	0.266090
12	uniform	2.580000	1000	0.000800	0.076000	0.209564	0.227528
12	uniform	2.580000	10000	0.000000	0.032280	0.205556	0.186452
12	uniform	3.000000	0	0.090900	0.118418	0.619996	0.628178
12	uniform	3.000000	10	0.078100	0.091373	0.161330	0.162278
12	uniform	3.000000	100	0.059420	0.353653	0.648856	0.641580
12	uniform	3.000000	1000	0.036800	0.209340	0.718166	0.668078
12	uniform	3.000000	10000	0.042000	0.088060	0.599304	0.567684
12	uniform	4.000000	0	0.090900	0.104604	0.824098	0.912858
12	uniform	4.000000	10	0.095200	0.104882	0.156544	0.176300
12	uniform	4.000000	100	0.087200	0.286560	0.698218	0.791000
12	uniform	4.000000	1000	0.090000	0.136500	0.815430	0.948864
12	uniform	4.000000	10000	0.096200	0.077380	0.771958	0.959398
12	uniform	optimal	0	0.410888	0.919648	0.999100	0.999900
12	uniform	optimal	10	0.135400	0.155000	0.141892	0.134000
12	uniform	optimal	100	0.463800	0.880324	0.703304	0.663918
12	uniform	optimal	1000	0.728600	0.986040	0.949050	0.815904
12	uniform	optimal	10000	0.752000	0.990000	0.985014	0.839900

Figure A-24: Pre-loading results for uniformly samples matrices with 12 states

States	Sampling	Beta	Pre-loading	10	100	1000	10000
12	$KL < 1$	2.000000	0	0.118372	0.170872	0.135756	0.121676
12	$KL < 1$	2.000000	10	0.059800	0.075480	0.066080	0.050252
12	$KL < 1$	2.000000	100	0.050600	0.087500	0.105682	0.107926
12	$KL < 1$	2.000000	1000	0.007600	0.049340	0.109388	0.107548
12	$KL < 1$	2.000000	10000	0.005600	0.025420	0.114564	0.107194
12	$KL < 1$	2.580000	0	0.103828	0.248972	0.269792	0.214066
12	$KL < 1$	2.580000	10	0.077600	0.106220	0.073548	0.091500
12	$KL < 1$	2.580000	100	0.030600	0.196640	0.229670	0.229898
12	$KL < 1$	2.580000	1000	0.009000	0.099460	0.189108	0.230466
12	$KL < 1$	2.580000	10000	0.004000	0.051460	0.214274	0.156880
12	$KL < 1$	3.000000	0	0.090900	0.126068	0.671278	0.574630
12	$KL < 1$	3.000000	10	0.078800	0.092100	0.151382	0.157156
12	$KL < 1$	3.000000	100	0.072000	0.338280	0.585968	0.671862
12	$KL < 1$	3.000000	1000	0.036800	0.226700	0.616530	0.683822
12	$KL < 1$	3.000000	10000	0.038000	0.111140	0.545466	0.553852
12	$KL < 1$	4.000000	0	0.090900	0.100020	0.821918	0.958378
12	$KL < 1$	4.000000	10	0.098000	0.124540	0.172054	0.197900
12	$KL < 1$	4.000000	100	0.078600	0.301700	0.679190	0.766694
12	$KL < 1$	4.000000	1000	0.090200	0.142660	0.820550	0.936934
12	$KL < 1$	4.000000	10000	0.090000	0.079980	0.794880	0.935226
12	$KL < 1$	optimal	0	0.394522	0.923612	0.999100	0.999900
12	$KL < 1$	optimal	10	0.109400	0.148780	0.145894	0.106000
12	$KL < 1$	optimal	100	0.418800	0.778160	0.705304	0.647930
12	$KL < 1$	optimal	1000	0.672600	0.978120	0.921078	0.751900
12	$KL < 1$	optimal	10000	0.759000	0.984060	0.981018	0.789902

Figure A-25: Pre-loading results for biased ($KL < 1$) matrices with 12 states

States	Sampling	Beta	Pre-loading	10	100	1000	10000
12	$KL < 0.75$	2.000000	0	0.116262	0.173410	0.119446	0.104144
12	$KL < 0.75$	2.000000	10	0.068600	0.071100	0.042360	0.062865
12	$KL < 0.75$	2.000000	100	0.029800	0.067200	0.110718	0.095908
12	$KL < 0.75$	2.000000	1000	0.012200	0.065600	0.111052	0.107800
12	$KL < 0.75$	2.000000	10000	0.004000	0.044780	0.100646	0.114298
12	$KL < 0.75$	2.580000	0	0.103424	0.201540	0.214810	0.177860
12	$KL < 0.75$	2.580000	10	0.065000	0.096540	0.089232	0.095314
12	$KL < 0.75$	2.580000	100	0.045600	0.138920	0.239994	0.259492
12	$KL < 0.75$	2.580000	1000	0.008000	0.105020	0.181554	0.198914
12	$KL < 0.75$	2.580000	10000	0.004167	0.054420	0.193440	0.160732
12	$KL < 0.75$	3.000000	0	0.091910	0.115550	0.515012	0.438779
12	$KL < 0.75$	3.000000	10	0.087500	0.108680	0.143908	0.141252
12	$KL < 0.75$	3.000000	100	0.078200	0.291940	0.534410	0.683016
12	$KL < 0.75$	3.000000	1000	0.014000	0.202640	0.446028	0.484368
12	$KL < 0.75$	3.000000	10000	0.002000	0.089540	0.375938	0.386760
12	$KL < 0.75$	4.000000	0	0.090900	0.099100	0.836116	0.891876
12	$KL < 0.75$	4.000000	10	0.100000	0.122680	0.138770	0.147364
12	$KL < 0.75$	4.000000	100	0.079800	0.315280	0.643542	0.655322
12	$KL < 0.75$	4.000000	1000	0.052000	0.264300	0.823764	0.893718
12	$KL < 0.75$	4.000000	10000	0.058000	0.148860	0.738292	0.908088
12	$KL < 0.75$	optimal	0	0.392702	0.794782	0.999100	0.999900
12	$KL < 0.75$	optimal	10	0.093400	0.129160	0.123920	0.129787
12	$KL < 0.75$	optimal	100	0.328000	0.748500	0.645362	0.561930
12	$KL < 0.75$	optimal	1000	0.636600	0.966240	0.895104	0.721900
12	$KL < 0.75$	optimal	10000	0.681600	0.984060	0.939060	0.761902

Figure A-26: Pre-loading results for biased ($KL < 0.75$) matrices with 12 states

States	Sampling	Beta	Pre-loading	10	100	1000	10000
18	uniform	2.000000	0	0.097465	0.304216	0.177306	0.149648
18	uniform	2.000000	10	0.077800	0.098160	0.112086	0.131044
18	uniform	2.000000	100	0.056000	0.149620	0.218750	0.191280
18	uniform	2.000000	1000	0.002000	0.062560	0.174644	0.169448
18	uniform	2.000000	10000	0.000000	0.013500	0.118332	0.142404
18	uniform	2.580000	0	0.094738	0.163928	0.359264	0.290382
18	uniform	2.580000	10	0.063000	0.094200	0.140668	0.153986
18	uniform	2.580000	100	0.047800	0.193300	0.323494	0.383002
18	uniform	2.580000	1000	0.009600	0.104480	0.290284	0.346366
18	uniform	2.580000	10000	0.004000	0.028800	0.261922	0.275616
18	uniform	3.000000	0	0.093526	0.122964	0.630270	0.573952
18	uniform	3.000000	10	0.092000	0.083300	0.138822	0.186576
18	uniform	3.000000	100	0.050600	0.252360	0.576166	0.511334
18	uniform	3.000000	1000	0.036200	0.149860	0.547884	0.651942
18	uniform	3.000000	10000	0.022000	0.087680	0.522602	0.628444
18	uniform	4.000000	0	0.092718	0.099380	0.816640	0.923900
18	uniform	4.000000	10	0.098000	0.108700	0.148618	0.198542
18	uniform	4.000000	100	0.085200	0.197860	0.596410	0.667832
18	uniform	4.000000	1000	0.082000	0.100760	0.756502	0.913258
18	uniform	4.000000	10000	0.084000	0.051580	0.749604	0.841196
18	uniform	optimal	0	0.436344	0.808656	0.999100	0.999900
18	uniform	optimal	10	0.091600	0.127200	0.177868	0.211998
18	uniform	optimal	100	0.325600	0.704920	0.687326	0.583926
18	uniform	optimal	1000	0.706400	0.990000	0.993006	0.861900
18	uniform	optimal	10000	0.760000	0.990000	0.999000	0.927900

Figure A-27: Pre-loading results for uniformly sampled matrices with 18 states

States	Sampling	Beta	Pre-loading	10	100	1000	10000
18	$KL < 1$	2.000000	0	0.096556	0.263370	0.159294	0.151450
18	$KL < 1$	2.000000	10	0.085400	0.095120	0.113040	0.140942
18	$KL < 1$	2.000000	100	0.039200	0.152120	0.201632	0.196012
18	$KL < 1$	2.000000	1000	0.005800	0.058220	0.167726	0.147904
18	$KL < 1$	2.000000	10000	0.002000	0.035700	0.147182	0.162414
18	$KL < 1$	2.580000	0	0.095546	0.184266	0.304450	0.261370
18	$KL < 1$	2.580000	10	0.074000	0.091100	0.104982	0.115454
18	$KL < 1$	2.580000	100	0.044800	0.168200	0.329114	0.362824
18	$KL < 1$	2.580000	1000	0.006000	0.133140	0.323840	0.286336
18	$KL < 1$	2.580000	10000	0.004000	0.046980	0.283662	0.249958
18	$KL < 1$	3.000000	0	0.094738	0.112370	0.651612	0.533888
18	$KL < 1$	3.000000	10	0.093200	0.083060	0.171138	0.159210
18	$KL < 1$	3.000000	100	0.060800	0.259360	0.490610	0.538672
18	$KL < 1$	3.000000	1000	0.048400	0.156980	0.608498	0.701626
18	$KL < 1$	3.000000	10000	0.041200	0.072540	0.480022	0.580286
18	$KL < 1$	4.000000	0	0.090900	0.102182	0.808106	0.946652
18	$KL < 1$	4.000000	10	0.096000	0.094380	0.141600	0.185264
18	$KL < 1$	4.000000	100	0.084400	0.202880	0.595882	0.620438
18	$KL < 1$	4.000000	1000	0.087200	0.116240	0.736148	0.895070
18	$KL < 1$	4.000000	10000	0.090000	0.052900	0.778376	0.875744
18	$KL < 1$	optimal	0	0.427256	0.850278	0.999100	0.999900
18	$KL < 1$	optimal	10	0.108400	0.135100	0.123926	0.169998
18	$KL < 1$	optimal	100	0.319000	0.695100	0.677332	0.553946
18	$KL < 1$	optimal	1000	0.677200	0.986040	0.997002	0.817900
18	$KL < 1$	optimal	10000	0.771600	0.990000	0.999000	0.913900

Figure A-28: Pre-loading results for biased ($KL < 1$) matrices with 18 states

Bibliography

[1] J P. Anderson. Computer Security Threat Monitoring and Surveillance. James P. Anderson Co, April 1980. Fort Washington PA.

[2] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Chalmers Univ., March 2000.

[3] M.J. Bach. *The Design of the Unix Operating System*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1986.

[4] David S. Bauer and Michael E. Koblentz. NIDX - An Expert System for Real-Time Network Intrusion Detection. pages 98–106. IEEE, April 1988. 11-13th, Washington, DC.

[5] M. Berthold and D. J. Hand. *Intelligent Data Analysis*. Springer, 2003.

[6] M. Ramoni — P. Sebastiani — P. Cohen. Bayesian clustering by dynamics. *Machine Learning*, 47(1):91–121, 2002.

[7] G. F. Cooper and E. Herskovits. A bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 1992.

[8] D. Denning. An intrusion-detection model. In *IEEE Transactions on Software Engineering*, 1987.

[9] Dorothy E. Denning. An Intrusion-Detection Model. 1986.

[10] Jon Doyle, Isaac Kohane, William Long, and Peter Szolovits. The architecture of MAITA - a tool for monitoring, analysis, and interpretation.

[11] William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1: Third edition. John Wiley and Sons, Inc., 1950.

[12] Stephane Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, 1997.

[13] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[14] P. Brinch Hansen. *Operating Systems Principles*. Series in Automatic Computation. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1973.

[15] G. Helmer, J. Wong, V. Honavar, and L. Miller. Intelligent agents for intrusion detection, 1998.

[16] Wenke Lee and Salvatore Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998.

[17] Wenke Lee, Salvatore Stolfo, and Patrick Chan. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of the AAAI97 workshop on AI methods in Fraud and risk management*, 1997.

[18] S.J. Leffler, M.K. Mc Kusik, M.J. Karels, and J.S. Quarterman. *4.3 BSD UNIX Operation System*. Addison-Wesley, Reading, Massachusetts, USA, 1989.

[19] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3), 1997.

[20] S. V. Raghavan and B. Balajinath. Intrusion detection through learning behavior model. *International Journal Of Computer Communications*, 2001.

[21] P. Sebastiani and M. Ramoni. Incremental bayesian segmentation of categorical temporal data. 2000.

[22] A.S. Tannenbaum. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1992.

[23] Christina Warrender, Stephanie Forrest, and Barak A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.