NORGES TEKNISK-NATURVITENSKAPELIGE UNIVERSITET

FAKULTET FOR INFORMASJONSTEKNOLOGI, MATEMATIKK OG ELEKTROTEKNIKK

HOVEDOPPGAVE

Kandidatens navn: Martin Thorsen Ranang

Fag: Datateknikk og informasjonsvitenskap

Oppgavens tittel (norsk):

Oppgavens tittel (engelsk): An Artificial Immune System Approach to Preserving Security in Computer Networks

Oppgavens tekst:

Biological systems, such as human beings, can be regarded as sophisticated information processing systems, and can be expected to provide inspiration for various ideas to science and engineering. One of the new branches of research, stemming from biological inspirations, is the are of artificial immune systems. An interesting area to investigate is how artificial life in general, and artificial immune systems in particular, may be used to solve real-world problems.

The project involves the implementation of a network intrusion detection system based on an architecture for artificial immune systems. The purpose is to explore how artificial immune systems may be used to preserve security in a computer network environment. Further, it will be examined how different phenomena that occur in the biological immune system, such as somatic hypermutation, will influence the features of the artificial model.

Oppgaven gitt:	20. januar 2002
Besvarelsen leveres innen:	24. juni 2002
Besvarelsen levert:	24. juni 2002
Utført ved:	Institutt for datateknikk og informasjonsvitenskap
Veileder:	Keith L. Downing

Trondheim, 24. juni 2002

Keith L. Downing
Faglærer

An Artificial Immune System Approach to Preserving Security in Computer Networks

Martin Thorsen Ranang

24TH JUNE 2002

This document was set in Palatino by the author using $\LaTeX\,2_\varepsilon$.

Contents

Illustrations

Tables

Preface

An Artificial Immune System Approach to Preserving Security in Computer Networks describes the thesis work performed by a 5th year student at the *Department of Computer and Information Science (IDI)* of the *Faculty of Information Technology, Mathematics and Electrical Engineering (IME)* at the *Norwegian University of Science and Technology (NTNU)*. This report is the result of an obligatory 10vt (weighting, where each vt corresponds to 4.8 hours of work per week) assignment in the 10th semester of the sivilingeniør (graduate degree in computer and information science) education. The advisor for the project has been Professor *Keith L. Downing*.

The field of artificial immune systems was presented to the author by Professor *Downing*, late in the autumn of 2001. The introduction was given in the form of a reference to an article (Hofmeyr and Forrest 2000). All other literature used for reference during this work has been sought and acquired by the author.

The initial task was to implement a prototypical artificial immune system applied to some aspect of computer security, with the presumption that there certainly would be some aspects of its workings that would be worth a closer look. The task description ("Oppgavens tekst") presented on the cover page is a refined description of this task, written by the author.

The purpose of the thesis work has been to gain a better insight into the research area of artificial immune systems in general, how it may be applied to areas such as security in computer network environments, and how a phenomenon taking place in the human immune system, called somatic hypermutation, may be modeled in an artificial immune system. The work also includes the implementation of an artificial immune system for network intrusion detection, called DAIS.

Martin Thorsen Ranang

Acknowledgments

I would like to thank my advisor, Professor *Downing*, for inspiring conversations and the helpful suggestions he has made during the thesis work. Even before the main thesis work has started, he showed an enthusiasm few others have.

Thanks to Professor *Stephanie Forrest* and *Justin Balthrop* at the *University of New Mexico (UNM)* Computer Science department for their support and for providing the simulation data used in the simulations in this thesis.

I would also like to thank *Anders Christensen* at IDI for his reflections on optimization and performance of the implementation of the simulation program. Tanks to *Rune Sætre* at IDI for sharing his ideas during the initial discussions concerning the statistics of the *contiguous-match* rule. Back then, we saw only the tip of the iceberg.

Thanks both to *Martin Kermit* and *Marte Fodstad* for being there, willing to listen to my problems and trying to help out. You gave me some important feedback during some of the most difficult parts of this work.

I am very thankful for the time Associate Professor *Harald Hanche-Olsen* at the *Department of Mathematical Sciences (IMF)* spent with me, discussing generating functions and power series.

Thanks to *Pavel Petrovic* for introducing me to the *Clustis* computational cluster. Thanks to *Zoran Constantinescu-Fülöp* for providing access to it, and for all his support. The cluster was used for running the simulations performed as part of this work. Those huge experiments would probably not have been possible to complete in time without such vast amounts of computing power.

Thanks to *Jo Henning Myhrvang* for his encouraging, inspiring and non-technical conversations during this project. Finally, I would like to thank *Ingunn Grytting*, my friends and my parents for all their patience, love and care.

Abbreviations

A-life	artificial life
AIS	artificial immune system
AMD	Advanced Micro Devices
ARPANET	Advanced Research Projects Agency Network
ARTIS	Artificial Immune System
ASCII	American standard code for information exchange
Ab	antibody
Ag	antigen
CDIS	Computer Defense Immune System
CERT/CC	CERT Coordination Center
CPU	central processing unit
CSMA/CD	carrier sense multiple access with collision detection
DAIS	Distributed Artificial Immune System
DARPA	Defense Advanced Research Projects Agency
DDOS	distributed denial of service
DNA	deoxyribonucleic acid
DOD	Department of Defense
EA	evolutionary algorithm
EC	evolutionary computation
EOI	event of interest
FTP	File Transfer Protocol
GAO	General Accounting Office
GB	gigabyte
GCC	GNU Compiler Collection
GNU	GNU's Not Unix!
GPL	General Public License
HD	hard drive
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
ID	intrusion detection
IDE	integrated device electronics
IDS	intrusion detection system
IMF	Department of Mathematical Sciences
IP	Internet Protocol
IPP	Internet Printing Protocol
IS	immune system

Ig	immunoglobulin
LAN	local area network
LGPL	Library General Public License
LISYS	Lightweight Intrusion Detection System
LRU	least recently used
LSB	least significant byte
MHC	major histocompatibility complex
MSB	most significant byte
MT	Mersenne Twister
NIC	network interface card
NID	network intrusion detection
NSF	National Science Foundation
OS	operating system
RAM	random access memory
RNA	ribonucleic acid
ROC	receiver operating curve or relative operating characteristic
SCSI	small computer system interface
TB	terabyte
TCP	Transmission Control Protocol
TCR	T-cell receptor
WWW	World Wide Web
XOR	exclusive OR
pH	process Homeostasis

Abstract

It is believed that many of the mechanisms present in the biological immune system are well suited for adoption to the field of computer intrusion detection, in the form of artificial immune systems. In this report mechanisms in the biological immune system are introduced, their parallels in artificial immune systems are presented, and how they may be applied to intrusion detection in a computer environment is discussed. An artificial immune system is designed, implemented and applied to detect intrusive behavior in real network data in a simulated network environment. The effect of costimulation and clonal proliferation combined with somatic hypermutation to perform affinity maturation of detectors in the artificial immune system is explored through experiments. An exact expression for the probability of a match between two randomly chosen strings using the *r*-contiguous matching rule is developed. The use of affinity maturation makes it possible to perform anomaly detection by using smaller sets of detectors with a high level of specificity while maintaining a high level of cover and diversity, which increases the number of true positives, while keeping a low level of false negatives.

CHAPTER 1

INTRODUCTION

While *artificial intelligence* uses the technology of computation as a model of intelligence, *artificial life (A-life)* is attempting to develop a new computational paradigm based on biological processes (Cho 2000). The field of artificial immune systems (AISs) is based on ideas and models that appear in biological immune systems (ISs), thus the field of AISs is a subfield of A-life.

The CERT Coordination Center (CERT/CC) has been observing computer intrusion activity since 1988 and reports that the number of calls and incidents reported to the CERT/CC has been almost doubling every year since they started.

The IS is a complex natural defense mechanism which, to some degree, is present in all living creatures. The IS has the ability to learn about foreign substances (*pathogens*) in the body and to invoke the correct response to fight the infectious agents. It is believed that the IS is a compelling real-world example of a massively parallel adaptive information processing system, as it exhibits many properties one would like to incorporate into artificial systems (Hofmeyr and Forrest 2000). AISs are computer programs that perform data manipulation, classification, reasoning and use representation methodologies that are based on the mechanisms in the human IS. It is therefore believed that many of the mechanisms present in the IS are well suited for adoption to the field of *intrusion detection (ID)*, in the form of AISs.

Presented in this report is the study of how AISs may be applied to computer *network intrusion detection*. Further, the implementation of an AIS, called Distributed Artificial Immune System (DAIS) is presented. The results of experiments performed with DAIS to explore the effect of *somatic hypermutation* in an AIS are presented and analyzed. It also presents the author's analysis of the probability of a match between two randomly chosen strings using the *r-contiguous match* rule, which results in a more exact expression for this probability than has been used in earlier research on AISs.

The rest of this report is structured as follows. In Chapter 2 an short introduction to the biological IS is given. The most important mechanisms, often modeled in AISs, are presented more thoroughly. Chapter 3 presents the field of *network intrusion detection (NID)*, with an emphasis on the central approaches and techniques used. Chapter 4 presents the field of AISs. Relevant work performed by others are mentioned and a presentation of the most relevant mechanisms of the IS as applied to AISs are presented. The author's analysis and development of an expression for the probability of a match using the *r*-contiguous match rule is presented. Chapter 5 presents the implementation of DAIS and how it models some important aspects of the IS. Chapter 6 presents the experiments performed to investigate the role of somatic hypermutation in an AIS applied to NID. It also presents the results found. Chapter 7 discuss and analyze the results found in Chapter 6 and further discuss the role of AISs in general, and DAIS in particular, applied to NID.

CHAPTER 2

BIOLOGICAL IMMUNE SYSTEMS

This chapter introduces the mechanisms present in the biological IS from which the field of AIS draws its inspiration. It should be noted that the exploration of how the biological IS works is still the focus of much research. Hence there is a possibility that the information presented will be proven incorrect over the course of time.

2.1 IMMUNITY

In biology, *immunity* is the ability of an organism to resist attacks by invasive foreign substances. Such a foreign substance is called a *pathogen* and is recognized by the IS as an *antigen (Ag)*. The Ag can be virtually any kind of large foreign molecule inside the body, including those contained in infective agents, snake and scorpion venom, food, and other cells and tissues from various species, including humans.

There are two kinds of immunity; namely, *non-specific, innate immunity* and *specific, acquired immunity*. The innate immunity comprises an individual's skin, the linings of the respiratory and gastrointestinal tracts, as well as some other protective factors. These immune mechanisms inhibit or kill a wide variety of microbes, irrespective of whether these have challenged the body before. It is non-specific in that its mechanisms can act against microbes that are not necessarily similar to one another.

After an individual has contracted a disease and recovered, it generally does not catch that illness again. This phenomenon is called acquired immunity. It is specific in that its responses are tailored to act against a particular microbe or its products. Such immunity is acquired in that the tailor-made responses are enormously increased as a result of being stimulated by the prior presence of a given microbe or its products.

To be able to resist attacks by Ags, the IS must be able to distinguish between the materials of the body and the materials of the foreign substance. As all living creatures are made up of basically similar building blocks, the ability of an organism to distinguish the molecules of which itself is composed—i.e., *self*—from practically all others—i.e., *nonself*—is remarkable. This ability is, to some degree, present in all living creatures, but among vertebrates it is especially a feature of the white blood cells called *lymphocytes*.

2.2 Lymphocytes and Antigens

Lymphocytes are the cells responsible for the body's ability to distinguish and react to an almost infinite number of different Ags. There are two main kinds of lymphocytes, i.e., *B-* and *T-lymphocytes* (also known as B- and T-cells). The stem cells for both B- and T-lymphocytes originate in the *bone marrow*. Recognition of foreign Ags in the IS is carried out by *receptors* on the surface of both the B- and T-lymphocytes.

The part of an Ag that is recognized by a *receptor*, the *antigenic determinant*, is called an *epitope* (Roitt and Delves 2001). Hence, an epitope is a location on the surface of a pathogen or a protein fragment, which is called a *peptide*. Many Ags have a variety of epitopes on different areas of their surface. Thus, complex Ags may invoke responses from a variety of specific *lymphocytes*.

Lymphocytes are mainly a dormant population, awaiting the appropriate signals to be stirred into actions. They move only sluggishly on their own, but they can be transported around the body, carried along in the blood or the lymph. At any one time, an adult person possesses about 2×10^{12} lymphocytes, about 1% of which are in the bloodstream.

2.2.1 B-Lymphocytes

The *B-lymphocytes*—also called *immunoglobulin (Ig)*—are differentiated in the *bone marrow* (hence the B). Each B-lymphocytes is programmed to make *antibody (Ab)* of a single specificity and place it on its outer surface to act as a *receptor*. Each B-lymphocyte has of the order of 10^5 identical Ab molecules on its surface.

When an Ag enters the body, it is confronted with a vast array of *lymphocytes*, all bearing different Abs each with its individual recognition site. The Ag will only bind to those receptors with which it makes a good fit. When the receptors of a lymphocyte have bound Ag, it receives a triggering signal and develop into Ab-forming plasma cells. Since the lymphocytes are programmed to produce one, and only one, kind of Ab, the Abs generated by the plasma cells will behave just like the Ab originally acting as a receptor; thus, it will bind well to the Ag (Roitt and Delves 2001, pp. 24, 25).

2.2.2 T-Lymphocytes

Many microorganisms, e.g. viruses, take advantage of the fact that if they live inside cells of their host, humoral Ab—as presented on the surface of *B-lymphocytes*—is unable to reach them. Thus, a *lymphocyte* subpopulation, comprising the *T-lymphocytes*, which is specialized to operate against intracellular organisms exists. Unlike the B-lymphocytes, the T-lymphocytes differentiate within the *thymus gland*. The *thymus* is a pyramid-shaped lymphoid organ. In humans it resides immediately beneath the breastbone at the level of the heart. The organ is called thymus because its shape resembles that of a thyme leaf.

The T-lymphocytes are different from the B-lymphocytes in that they only recognize Ag when it is presented on the surface of a body cell. Therefore, the *T-cell receptors (TCRs)*, which are different from the Ab molecules used by the B-lymphocytes, recognize Ag *plus* a surface marker indicating that it is presented on the surface of another cell. These cell markers belong to a group of molecules called the *major histocompatibility complex (MHC)*[1].

[1]The term histocompatibility was derived from the Greek word *histo* (meaning "tissue") and the English word compatibility due to the MHC's ability to evoke powerful transplantation reactions.

Each individual in a population is genetically capable of making a small set of these MHC markers (about 10), but the set of MHC types varies from one individual to another. Hence, individuals in a population are capable of recognizing different profiles of *peptides*. This mechanism provides an important form of population-level *diversity* (Hofmeyr and Forrest 2000).

2.2.3 Recognition of Antigens

The genetic rearrangement described below only takes place when the *lymphocytes* first become functional.

The binding of an Ab to an Ag, or of a TCR to a MHC-*peptide* complex, requires that portions of the two structures have complementary shapes that can closely approach each other (Percus, Percus, and Perelson 1993). This complementarity of shape allows the *receptor* and the Ag to conform to each other in a fashion roughly analogous to the way a key fits into a lock. This matching is approximate, which enables one particular lymphocyte to bind to several different kinds of structurally related *pathogens*.

The area on an Ag where it has contact with an Ab is, as mentioned in Section 2.2, called an *epitope*. The corresponding area on an Ab is called a *paratope*. The strength of the binding of an Ag to a single Ab combining site is dependent of the *affinity* between them. The higher the affinity, the stronger the binding.

It is known that the IS is capable of recognizing virtually any pathogen that exists or that may be devised either by nature or by science in the future. To accomplish this task the IS generates millions of different specific Ag receptors, which is probably vastly more than is needed during an individuals lifetime (Roitt and Delves 2001).

Ag receptor molecules are proteins that are composed of a few *polypeptide* chains. A polypeptide is a peptide containing from 10 to more than 100 amino acids. The sequence in which the amino acids are assembled to form a particular polypeptide chain is designated by the genes of the *deoxyribonucleic acid (DNA)*. Since the entire human genome only contains about 30 000 to 40 000 genes (Venter et al. 2001), individuals cannot inherit a single gene for each particular Ag receptor site. Therefore, a limited pool of gene segments is inherited for each type of polypeptide chain, which code for Abs and TCRs. As each lymphocyte matures, the gene segments are pieced together to form one gene for each polypeptide that makes up a specific receptor. This gene segment rearrangement, for the most part, occurs at random.

The Ab and TCR repertoire in a mouse is estimated to contain on the order of 10^7 different receptors generated from a much larger potential repertoire of germ-line-encoded receptors (Percus, Percus, and Perelson 1993).

2.3 Negative Selection

When *T-lymphocyte* precursors leave the *bone marrow* on their way to mature in the *thymus*, they are indifferent to stimulation by Ag as they do not yet express *receptors*. When they enter the thymus they are called *immature lymphocytes* and when, or if, they leave they are called *mature lymphocytes*. This maturation process is often called *tolerization*. Within the thymus the T-lymphocytes multiply many times as they pass through a meshwork of thymus cells. As they multiply, they acquire receptors and differentiate into different T-lymphocyte subclasses.

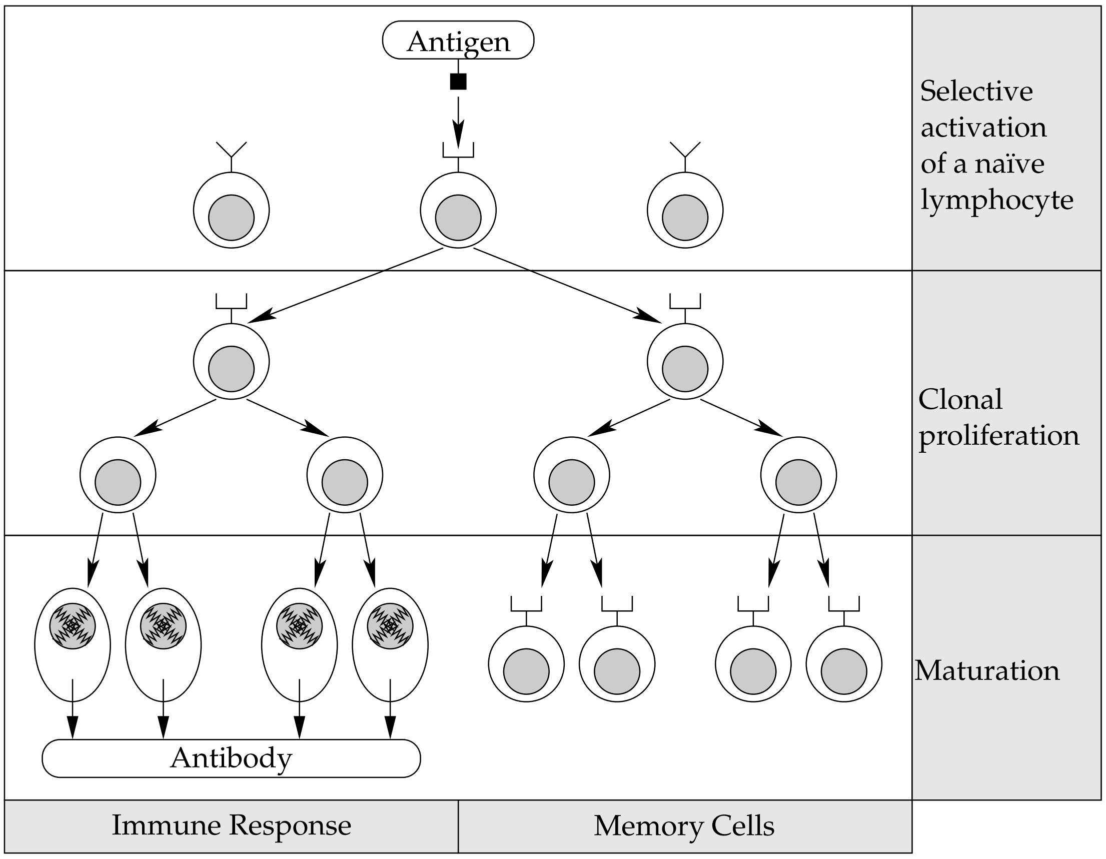


FIGURE 2.1: The cellular basis for the generation of effector and memory cells by clonal selection after primary contact with antigen. Based on a figure from (Roitt and Delves 2001, p. 27).

In the body, of the immature T-lymphocytes entering the thymus, only 2% complete the maturation process and become functioning T-lymphocytes (Forrest, Hofmeyr, and Somayaji 1997). This means that most of the T-lymphocytes entering the thymus also die there, during tolerization. This may seem very wasteful, but as the Ag receptors are randomly created, a lot of them will recognize *self* Ags. Self Ags are molecules present on the body's own constituents. If *lymphocytes* which are *autoreactive*—i.e., they react to self—become mature they will attack the body's own tissues. Therefore most of them are deleted by *apoptosis* in the thymus. Apoptosis is a kind of programmed cell death. This mechanism for preventing the development of *autoimmune lymphocytes* is called *negative selection*. Negative selection of developing *B-lymphocytes* is also thought to occur if they encounter high levels of self Ag in the bone marrow (Roitt and Delves 2001, p. 231).

2.4 Clonal Selection

The first encounter between a *naïve lymphocyte* and a given Ag is called a *primary immune response*. This response is relatively weak, compared to the *secondary immune response*, which is a qualitatively and quantitatively improved response that occurs upon the second encounter of *primed lymphocytes* with a given Ag (Roitt and Delves 2001, pp. 28–30). Part of this improved response is due to a process called clonal selection, which occurs after a *lymphocyte* has recognized a specific Ag. The process is shown in Figure 2.1 on the preceding page, where the lymphocyte selected for by a specific Ag undergoes many divisions during the clonal proliferation and the offspring mature to form an expanded population of Ab-forming cells. A fraction of the offspring of the original Ag-reactive lymphocytes becomes non-dividing *memory cells*.

The *secondary response* in the IS is stronger than the *primary response*. Since there are more *lymphocytes* that possibly bind to the Ag during the secondary response, the *infectious agent* will be defeated faster. Because of this mechanism, where the first contact with an Ag clearly imprints some information, or imparts some *memory*, into the IS it is said that the IS develops an *acquired memory* (Roitt and Delves 2001, p. 28). The memory induced by one Ag will not automatically extend to another unrelated Ag.

During the *clonal proliferation* there is an exponentially growing population of lymphocytes that are able to detect the activating Ag. The lymphocytes with the highest *affinity* between its *receptors* and the *pathogen epitopes* are most likely to be activated later on. The pathogens are usually also replicating during this time period, so there is a race to become the strongest population.

2.5 Somatic Hypermutation

During a *primary response*, *B-lymphocytes*—but generally not *T-lymphocytes*—undergo high rate point mutation in their variable region genes. This mechanism is called *somatic hypermutation* and increases Ab *diversity* and Ab *affinity*. It is called somatic[2] because it takes places in body cells rather than in germ-line cells (eggs and sperm). The mutations are the result of single *nucleotide* substitutions and are restricted to the variable region of the *lymphocytes*, which means that the constant regions are not affected by these mutations. A nucleotide is the basic structural unit of nucleic acids, such as the DNA or the *ribonucleic acid (RNA)*. The sequence of nucleotides in the DNA or RNA codes for the structure of proteins synthesized in the cell.

The *mutation rate* during somatic hypermutation is approximately 10^{-4}–10^{-3} per base pair per generation, which is approximately a million times higher than for other mammalian genes (Roitt and Delves 2001, pp. 68, 69).

It is believed that somatic hypermutation is a way for the IS of increasing its chances in the "proliferation race" mentioned in Section 2.4 above. The somatic hypermutation combined with clonal expansion is an adaptive process known as *affinity maturation* (Hofmeyr and Forrest 2000).

2.6 Costimulation

Because some *self peptides* are never expressed in the *thymus*, mature *lymphocytes* that have been *tolerized* in the thymus may bind to these proteins and cause an autoimmune reaction (Hofmeyr and Forrest 2000, p. 450). In practice this does not happen because in addition to binding to an Ag, a *T-lymphocyte* needs to receive a *costimulation* signal in

[2]The Greek word *soma* means body.

order to be activated. This signal is usually some kind of a chemical signal that is produced when the body is damaged in some way.

Chapter 3

Network Intrusion Detection

The *Internet* has its origin in a U.S. *Department of Defense (DOD)* program called *Advanced Research Projects Agency Network (ARPANET)* that was established in 1969 to provide a secure communications network for organizations engaged in defense-related research. Researchers and academics in other fields began to make use of the network. The *National Science Foundation (NSF)* took over much of the technology from ARPANET and established a distributed network of networks capable of handling far greater traffic, which is today known as the Internet.

From its creation, the *Internet* grew rapidly beyond its largely academic origin into an increasingly commercial and popular medium. The original use of the *Internet* was to send and receive e-mail, to transfer files, to visit bulletin boards and newsgroups and to access remote computers (by using *telnet*). The *World Wide Web (WWW)*, which enables simple and intuitive navigation of hypertext at Internet sites through a graphical interface, expanded dramatically during the 1990s to become the most important component of the *Internet*. With the popularity of the *Internet* a lot of new individuals have become members of the *Internet user community*, and as in most other communities, some individuals have good intentions while some have bad intentions.

The CERT/CC[1] has been observing computer intrusion activity since they were founded in 1988. The graph in Figure 3.1 on the following page shows the trends in the statistics presented in (CERT Coordination Center 2002). The same information is presented numerically in Table 3.1 on page 11. The statistics show how the calls to and incidents reported to CERT/CC has roughly been doubling every year, since 1988. At the same time, it is reported by Householder, Houle, and Dougherty (2002) that the level of automation in attack tools and their sophistication continues to increase. It is also reported that while *Internet firewalls* are often relied upon to provide an organizations first line of defense against intruders, technologies are being designed to bypass typical firewall configurations. Such technologies includes the *Internet Printing Protocol (IPP)*, a protocol designed to cover the most common situations for printing on the *Internet*, which in itself is a non-intrusive activity, but when it bypasses firewall configurations, it becomes a security hazard. Certain aspects of mobile

[1] *CERT* was originally called the computer emergency response team.

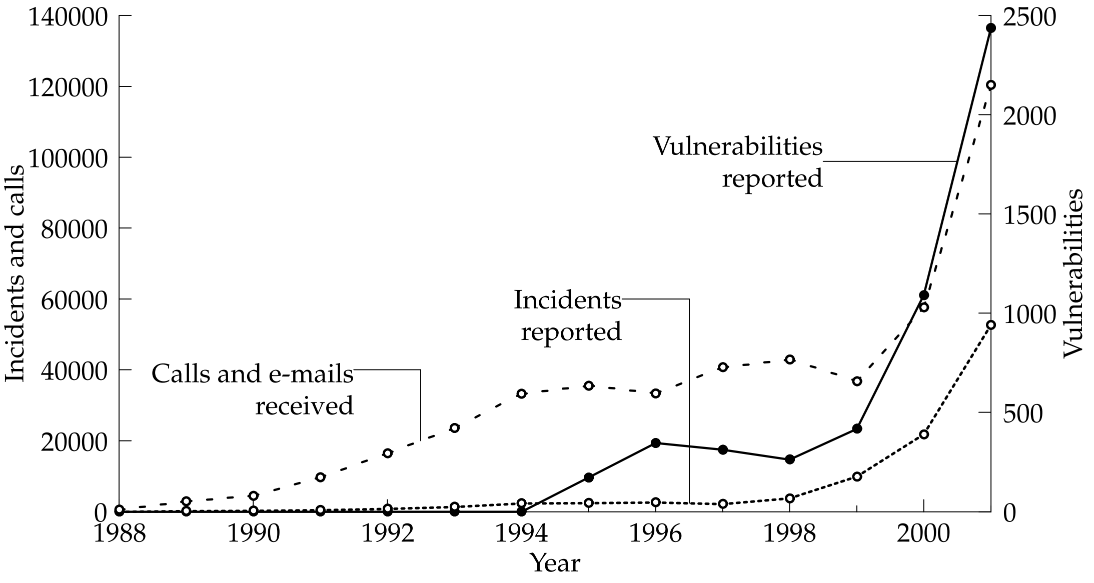


FIGURE 3.1: Trends in CERT/CC statistics. Be aware that the curves for "Calls and e-mails received" and "Incidents reported" refer to the left y-axis, while the "Vulnerabilities reported" curve refers to the right y-axis. Permission to use the information, from (CERT Coordination Center 2002), was granted by the CERT/CC.

code make malicious software difficult to discover.

3.1 Intrusion Detection

The field of ID attempts to detect inappropriate, incorrect, or anomalous activity in a computer network, or on a host, by the exploration of certain kinds of data. Such activities may be initiated from external crackers or from internal misuse. In the following an activity of this kind will be referred to as an *event of interest (EOI)*. Note that an EOI does not necessarily have to be a deliberate *attack* by an external individual, but may rather be an effect of a legitimate user's anomalous actions or errors. Computer programs that perform ID are in general called *intrusion detection systems (IDSs)*.

3.2 Intrusion Detection Approaches

An IDS which operates on a computer to detect malicious activity *on that host*, is called a *host-based IDS*, while an IDS which tries to detect *events of interest* by analyzing and monitoring network traffic data is called a *network-based IDS* or a NID system. These are the two main approaches to *ID*. A host-based IDS will typically monitor and analyze different data input from users and output from the *operating system (OS)*. It may, for example, monitor *system-calls*. As stated by Stillerman, Marceau, and Stillman (1999), "The system calls are a side effect of the attack, just as fingerprints are a side effect of a burglar's presence."

Ideally, an organization would combine both of the above approaches to increase the probability of detecting intrusions.

TABLE 3.1: Number of incidents reported to the CERT/CC. Permission to use this information was granted by the CERT/CC.

Year	Incidents Reported	Vulnerabilities Reported	Mail Messages Handled	Phone Calls Received
1988	6	—	539	—
1989	132	—	2 869	—
1990	252	—	4 448	—
1991	406	—	9 629	—
1992	773	—	14 463	1 995
1993	1 334	—	21 267	2 282
1994	2 340	—	29 580	3 665
1995	2 412	171	32 084	3 428
1996	2 573	345	31 268	2 062
1997	2 134	311	39 626	1 058
1998	3 734	262	41 871	1 001
1999	9 859	417	34 612	2 099
2000	21 756	1 090	56 365	1 280+
2001	52 658	2 437	118 907	1 417+
Total	100 369	5 033	437 528	20 287+

3.3 INTRUSION DETECTION TECHNIQUES

The two basic categories of techniques used to perform ID—both in *host-based* and *network-based ID*—are *anomaly-based detection* and *signature-based detection*. The latter technique is also known as *misuse detection* and *pattern-matching detection*.

McHugh, Christie, and Allen (2000) compare the task of ID to a general signal-detection problem. In this case, one would view an EOI as the signal to be detected and normal behavior as the noise. In classical signal-detection approaches some information is known about both the signal and the noise distributions, and a decision process uses information about both distributions to determine whether a given observation belongs to the signal-plus-noise distribution or to the noise distribution. In ID, on the other hand, the detectors typically base their decisions either on signal (*signature-based detection*) or noise (*anomaly-based detection*) characterizations. Each approach has its strengths and weaknesses, while both suffer from the difficulty of characterizing the distributions.

To successfully detect an *EOI*, a signature-based IDS relies on possessing a description, or signature, to be matched against the event. This signature can be as simple as a part of a network packet, and as complex as a neural network description that maps multiple sensors in an OS to an abstract attack representation. If an appropriate abstraction is used for creating the signature, a signature-based IDS can detect intrusions not previously seen if they are abstractly equivalent to known signatures.

The anomaly-based IDSs are based on the assumption that *unusual* or *abnormal* behavior is intrusive. By using the above comparison between ID and signal-detection, a anomaly-based IDS will detect intrusions whenever the observation does not appear to be noise alone, given that there exists a complete characterization of the noise distribution. It should be

TABLE 3.2: The relationship between *true/false negatives* and *positives* and *events of interest*.

Event of Interest	Detection	Classification
No	No	True negative
No	Yes	False positive
Yes	No	False negative
Yes	Yes	True positive

noted that characterizing the noise distribution so as to support detection is non-trivial.

3.4 Limits of Observation

There may be several reasons for a IDS not to be able to *observe*—and therefore not detect—an EOI. According to Northcutt, Novak, and McLachlan (2000, p. 147 f.), the reasons include that an EOI may occur on a different network than the one being under surveillance; the EOI may occur right in front of the IDS, but the system does not detect it because it is not operative; the EOI occurs in a protocol which the IDS does not understand; or that the EOI happens during a period when the bandwidth of the IDS is exceeded.

3.5 Negatives and Positives, True and False

When an IDS monitors and analyzes network data, its decision in each case falls into one of the categories shown in Table 3.2. A *false negative* occurs when an actual intrusive action has occurred but the IDS allows it to pass as non-intrusive behavior. A *false positive* occurs when the system classifies an action as anomalous—a possible intrusion—when it is a legitimate action (Northcutt et al. 2001, pp. 20, 21). A *true negative* occurs when there is no EOI and no detection is done. Finally, a *true positive* occurs when the IDS correctly classifies an EOI as intrusive behavior.

It should be quite straightforward to see that, from a user's perspective, one would like to minimize the number of false positives and maximize the number of true positives performed by an IDS. This would result in a low number of false alarms and a high number of intrusions detected, respectively. A high number of false alarms is unattractive because it may result in making the users of such a system off guard. Naturally, the true and false negatives are never noticed in a real-life system.

3.6 Local Area Networks

Halsall (1996) reports that the most widely installed type of *local area network (LAN)* is that based on the *carrier sense multiple access with collision detection (CSMA/CD)* access control. CSMA/CD is more commonly known as *Ethernet* which use the *Internet Protocol (IP)* as its main protocol. The IP[2] does not have any mechanisms to augment end-to-end data reliability, flow control, sequencing, or other services commonly found in host-to-host protocols (Postel 1981a), therefore it is often accompanied by the use of Transmission Control

[2]The IP is defined in (Postel 1981a).

Protocol (TCP)[3], which, according to (Postel 1981b, p. 1), is "intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks."

The trend has long been that organizations such as large corporations and universities have used *broadcast network* technology for their LANs. This trend is changing slightly, as many organizations have replaced their broadcast networks with *switched networks*. The main difference between broadcast LANs and switched LANs is that in a broadcast LAN, the network traffic addressed for *one* host will be pass by the *network interface card (NIC)* of *every other* host on the same LAN, while this is *not possible*[4] in a switched LAN. In a switched LAN every host is connected to a *switch* through in own dedicated physical network segment (cable).

[3]The TCP is defined in (Postel 1981b).

[4]Actually, it is *possible* (without hardware modifications), but only after some non-trivial hacking.

CHAPTER 4

ARTIFICIAL IMMUNE SYSTEMS

This chapter explores how central mechanisms of the biological IS may be abstracted and used to form an AIS. First, a short review of some of the related work on AISs is presented, then some important ideas of AISs are reviewed and discussed, while analogies of the AIS and the IS are emphasized. The main focus will be on mechanisms used in AISs modeled for use in ID.

4.1 RELATED WORK

As stated in Chapter 2, the ability of the *IS* to distinguish between *self* and *nonself* is remarkable. This central mechanism of the IS is often modeled in AISs as various kinds of *anomaly-based detection*. It should be noted that the IS is much more than a simple anomaly-based detection and response system; it can be viewed as a general pattern-learning system that is highly distributed and scalable.

The pattern-learning abilities of the IS has been modeled and described by Timmis, Neal, and Hunt (2000) and Dasgupta, Cao, and Yang (1999) who successfully applied their AISs to recognition and classification tasks. They showed that their IS-inspired models were flexible, noise-tolerant and generalized their classification well. It has also been shown that it is possible to perform these tasks effectively with resource limited AISs (Timmis and Neal 2001). The behavior of the IS from an information processing perspective is described by Forrest and Hofmeyr (2001).

It seems that the area where the notion of an AIS has been most widespread, is in the area of *computer security*. This idea has been explored in general by Somayaji, Hofmeyr, and Forrest (1998), Burgess (1998), Dasgupta (1999), Hofmeyr (1999) and Hofmeyr and Forrest (2000). This work has focused both on *computer security architectures* as a whole and on specific areas of computer security. The areas in computer security where AISs have been used, at least experimentally, range from *virus detection* (Forrest et al. 1994), as a kind of *host-based anomaly detection*, and OS process monitoring (Forrest et al. 1996) to NID (Hofmeyr and Forrest 2000; Williams et al. 2001), as a kind of distributed *network-based anomaly detection*.

Some *prototypical* NID systems have been implemented, such as *Computer Defense Immune System (CDIS)* by (Williams et al. 2001) and *Lightweight Intrusion Detection System (LISYS)* by Hofmeyr (1999) and Hofmeyr and Forrest (2000).

4.2 Constituents of Artificial Immune Systems

In the following, the theory on AIS mechanisms implemented in the AIS-based NID program called DAIS, described in Chapter 5, will be presented. The author will also develop a more exact/correct expression for the probability for a match between two randomly chosen strings using the *r-contiguous match* rule, which is an important constituent of much work on AISs. At the end of the chapter, some other approaches at modeling features of the IS will also be mentioned.

As the field of AISs is still quite new, it is difficult to present one coherent theory on AISs. On the other hand, it should be relatively easy to see how the different approaches to modeling AISs—and specific features of these models—have been inspired by the biological IS.

4.3 Detectors and Peptides

In the literature on AISs applied to ID the modeling of Abs and *lymphocytes* are often merged into the common entity of a *detector*. This practice is also used in the implementation of DAIS.

The ability to discriminate between *self* and *nonself* is probably the most fundamental feature of the IS. This is done through the recognition by lymphocytes of distinct Ags. As the recognition of Ag in the biological IS happens when chemical bonds are established between the *receptors* on the surface of immune cells and *epitopes* on the surface of *pathogens*, the matching at a low level comes down to matching of proteins or fragments of protein, which are called *peptides*. In the following, the word peptide will be used for the representation of both *artificial receptors* and artificial Ags.

In AISs peptides are often represented as strings of length ℓ, consisting of symbols from an alphabet of cardinality m. This approach was used with $m = 2$ (i.e., bit strings) and $\ell = 32$ in (Forrest et al. 1994), and with $\ell = 49$ in (Hofmeyr 1999; Hofmeyr and Forrest 2000). The same approach is adopted, with $l = 49$, in the implementation of DAIS.

The peptides representing the Ag will encode some information relevant to the problem domain, to which the AIS is applied. While the IS must distinguish between self and nonself based on peptides, the AIS must discriminate between self and nonself based on strings of a fixed length ℓ. Each such string will be referred to as an *agent* a. The set of all agents form a universe, $U = \{a_1, a_2, \ldots, a_n\}$ which comprises two disjoint subsets; i.e., the set of self, U_S, and the set of nonself, U_N, so that $U = U_S \cup U_N$ and $U_S \cap U_N = \emptyset$. As stated in (Hofmeyr and Forrest 2000), the AIS then face a classification problem; given an arbitrary string from U, classify it as either self or nonself. The classification of self and nonself may also be viewed as the discrimination between *normal* and *anomalous*.

This model of peptides adheres to the requirement that all relevant information in the problem domain can be represented in some way and that there must be some way of compactly encoding generalizations of this information.

$$
\begin{aligned}
a &= \texttt{110}\boxed{\texttt{01010}}\texttt{10}\boxed{\texttt{1}}\texttt{1}\boxed{\texttt{101}}\texttt{0} \\
b &= \texttt{001}\boxed{\texttt{01010}}\texttt{01}\boxed{\texttt{1}}\texttt{0}\boxed{\texttt{101}}\texttt{1}
\end{aligned}
$$

FIGURE 4.1: Matching under the Hamming match rule, with strings of length $\ell = 16$ consisting of symbols from an alphabet with cardinality $m = 2$, with the matching constraint $r = 9$. The two strings a and b will match for all $r \leq 9$.

It should also be noted that when real-world problems are mapped to representations like this, self and nonself may not be disjoint, because two cases may be mapped to the same representation.

Just as an IDS, the IS can also make two kinds of discrimination errors (see Table 3.2 on page 12). This is true also for the AIS. A *false positive* occurs when a normal *agent* is classified as *nonself* and a false negative occurs when an anomalous agent is classified as *self*.

The *peptide* encoding described above, is also used for modeling the *receptors* of *detectors* in the AIS. The AIS has a population D of detectors. Each detector $d \in D$ has a *cover* C_d that describes the number of agents it recognizes. If a detector d recognizes *no* agents, its cover is $C_d = \emptyset$. On the other hand, if d recognizes all other agents, its cover is $C_d = U$; all agents in the universe of discourse.

This representation of peptides enables the AIS to recognize different agents through string matching. But, as noted in Section 2.2.3, one of the nice features of the IS, seen from an information processing point of view, is that it is able generalize its matching. The generalization of self and nonself which occurs in the *IS* is implemented using *approximate string matching*.

4.3.1 MATCHING

In the most general form, the problem of *approximate string matching* is to find a text where a given text pattern occurs, allowing a limited number of "errors" in the matches. Each application uses a different error model, which defines how different strings may be (Navarro 2001). These texts may be regarded as sequences of symbols composed from an alphabet of cardinality m.

What matching rule to use depends on what characteristics one would like. Two such rules are the *Hamming match* rule and the *r-contiguous match* rule. Herein, the main focus will be on the r-contiguous match rule, because it is a plausible abstraction of the receptor binding in the immune system (Percus, Percus, and Perelson 1993). As seen in Section 2.2.3, the IS is very effective in that it manages to discriminate between *self* and *nonself*, with a relatively small *detector* repertoire. Using this rule for predicting the optimal size of the T-cell and Ab combining region from consideration of efficient self-nonself discrimination, the results were consistent with various experimental determinations on the number of contact residues between Ab combining sites and protein Ags and the size of the MHC-peptide complex that interacts with the TCR (Percus, Percus, and Perelson 1993).

Both the Hamming match rule and the r-contiguous match rule are controlled by a threshold parameter r, where $0 \leq r \leq \ell$. If $r = 0$, the cover of d is all strings, $C_d = U$ and if $r = \ell$, then the cover of d is a single *agent* string, $C_d = \{a\}$. The higher the value of r, the more specific the match. And, the specificity of a match is analogous to the *affinity* of a binding between an Ag and a *lymphocyte*, or detector.

$$a = \texttt{110}\boxed{\texttt{01010}}\texttt{10111010}$$
$$b = \texttt{001}\boxed{\texttt{01010}}\texttt{01101011}$$

FIGURE 4.2: A match under the r-contiguous match rule, with strings of length $\ell = 16$ consisting of symbols from an alphabet with cardinality $m = 2$, with the matching constraint $r = 5$. The strings a and b in the above example will match for all $r \leq 5$.

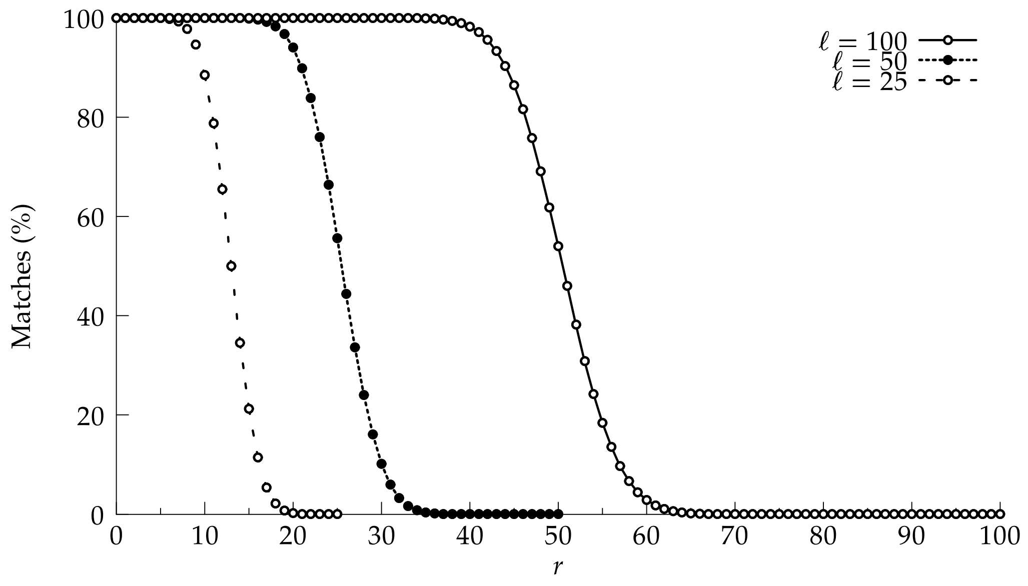


FIGURE 4.3: How the values of r and ℓ affects the probability of a match, using the Hamming match rule, when $\ell \in \{25, 50, 100\}$ and $0 \leq r \leq \ell$.

It should be noted that in the IS, the match (or recognition) between an Ag and a lymphocyte is based on complementary shapes, while in the following the discussion will be centered around binary strings ($m = 2$) and if these are approximately equal or not. The assumption is simply that in an *artificial peptide* match, a `1` on the *epitope* is complementary to a `1` on the *paratope*.

The *Hamming match* rule is based on the Hamming distance between two strings. If the two strings a and b have the same bits in at least r positions, they match. This is shown in Figure 4.1 on the preceding page.

Under the r-contiguous match rule, two strings match if the two strings a and b have the same bits in at least r *contiguous* positions. Figure 4.2 shows an example of this.

Probability of Match Using the Hamming Match Rule

Let $\text{Hamming}_{\ell,r}(a, b)$ be the operator to determine if two strings a and b, both of length ℓ, match, using the Hamming match rule, with the constraint that r bits are pairwise equal. Then, the probability of a match between two randomly chosen strings a and b, is

$$P(\text{Hamming}_{\ell,r}(a, b)) = 2^{-\ell} \sum_{i=r}^{\ell} \binom{\ell}{i} \tag{4.1}$$

as presented in (Hofmeyr 1999). The probability is derived by noting that $2^{-\ell}$ is the probability of a single match occurring, and $\binom{\ell}{i}$ is the number of strings in U that have the same bits in i positions.

Figure 4.3 on the facing page shows how the choices of r and ℓ affect the probability of a match between randomly chosen bit strings of length ℓ.

Probability of Match Using the Contiguous Match Rule

The texts on AISs that the author is aware of which rely on the *r-contiguous match* rule, all refer to the equation

$$P_S = m^{-r}\left(\frac{(\ell - r)(m-1)}{m} + 1\right) \tag{4.2}$$

presented in (Percus, Percus, and Perelson 1993), where P_S denotes the an approximation of the probability that a random *receptor* recognizes a randomly chosen Ag, abstracted as strings of a fixed length ℓ consisting of symbols from an alphabet with cardinality m, with a matching threshold r, using the r-contiguous match. The reasoning behind the formula, as given by Percus, Percus, and Perelson (1993) is that

> Denote a matching or a complementary pair by the symbol x and the non-complementation by y. If receptor and antigen are each constructed with the m units chosen at random, then at each position complementation occurs with the probability $1/m$ and noncomplementation occurs with the probability $(m-1)/m$. The probability of recognition P_S then translates into the problem of at least one sequence of at least r contiguous xs out of a total of ℓ entries. . . .
>
> . . . A rigorous analysis shows that the probability of a long matching region is very small, and hence when $m^{-r} \ll 1$, to a good approximation the various contributing possibilities can be regarded as independent. Starting at the leftmost site of the ℓ-site sequence, r contiguous xs occur with probability m^{-r}. Thereafter, runs of r xs can start at $\ell - r$ possible sites. Each such run is preceded by a mismatch y, for a net probability of $m^{-r}(m-1)/m$.

They arrived at the expression in (4.2) by adding up those probabilities. By substituting $m = 2$, $l = 49$ and $r = 4$, (4.2) evaluates to

$$P_S = 2^{-4}\left(\frac{(49-4)(2-1)}{2} + 1\right) = \frac{1}{16} \cdot \frac{47}{2} = 1.468\,75.$$

Hence, this example shows that P_S does not describe a probability distribution.[1]

Below, the author will present the analysis necessary to arrive at an exact expression for the probability of a match between two randomly chosen strings of a fixed length ℓ using the r-contiguous match rule.

Repeated independent trials are called *Bernoulli trials*[2] if there are only two possible outcomes for each trial and their probabilities remain the same throughout the trials (Feller

[1]The reason that $\ell = 49$ will become clear in Section 5.3. It is the length of the peptides used in the later experiments.

[2]Named after *Jacob Bernoulli* (Born in Basel, Switzerland on December 27, 1654 and died on August 16, 1705 in Basel). He is also referred to as Jaques or James.

1968a, p. 146). The two possible outcomes are usually assigned probabilities p for "success" and q for "failure". Both p and q must be non-negative, and

$$p + q = 1. \tag{4.3}$$

Since the trails are independent, their probabilities multiply. Thus, if S denote a success and F denote a failure, then $P(SFS \dots SS) = pqp \dots pp$.

According to Muselli (1996), the framework presented by Feller (1968b) for the study of *success runs* in *Bernoulli trials* has been used in most recent studies of the phenomenon. In particular it is the definition of a *run* as a *recurrent pattern* that has been adopted. The formal definition of success runs is:

> A sequence of n letters S and F contains as many S-runs of length r as there are non-overlapping uninterrupted blocks containing exactly r letters S each (Feller 1968b, p. 305).

According to this definition, two consecutive success runs may not be separated by any failure. Thus, the sequence $SSSSFSSSSSSF$ can be interpreted as containing 3 success runs of length 3 (occurring at trials number 3, 8, and 11). It may also be interpreted as containing 2 success runs of length 4 (occurring at trials number 4 and 9). In practice, if one is searching for runs of length r, the counting of consecutive successes must be restarted when the desired value of r is reached.

In the following, let r be fixed positive integer and let ε denote the occurrence of a success run, as defined above, of length r, in a sequence of Bernoulli trials; thus a run is a *recurrent event*. Further, let u_n be the probability of ε at the nth trial, and f_n be the probability that the *first* success run of length r occurs at the nth trial. For convenience, the variables $f_0 = 0$ and $u_0 = 1$ are also defined.

The probability that the r trials number $n, n-1, n-2, \dots, n-r+1$ result in success is p^r. In this case, ε occurs at one among these r trials. Hence, the probability that ε occurs at trial number $n-k$, where $k = 0, 1, \dots, r-1$, and the following k trials result in k successes equals $u_{n-k}p^k$. Feller (1968b, p. 323) states that since these r possibilities are mutually exclusive, one gets the *recurrence relation*

$$u_n + u_{n-1}p + \cdots + u_{n-r+1}p^{r-1} = p^r \tag{4.4}$$

valid for $n \geq r$, where $u_1 = u_2 = \cdots = u_{r-1} = 0$ and $u_0 = 1$.

It would be convenient to express the recurrence relation in (4.4) as a *generating function*. Wilf (1993, p. 1) writes that "a generating function is a clothesline on which we hang up a sequence of numbers for display", while *Abraham de Moivre*[3] introduced generating functions in order to solve the general linear recurrence problem. There are several things one may be able to do with generating functions, but the reason it is introduced here, is that it often enables one to find an exact formula for the members of a sequence. A generating function, $G(x)$, defined as

$$G(x) = a_0 + a_1x + a_2x^2 + \cdots = \sum_{n=0}^{\infty} a_n x^n \tag{4.5}$$

[3]Born 26 May 1667 in Vitry, France and died 27 November 1754 in London, England.

is a single quantity that represents the whole sequence $\{a_n\}$. As Knuth (1997a, p. 92) writes when introducing generating functions, it is often convenient to let the notation

$$[x^n]G(x) \tag{4.6}$$

denote the coefficient of x^n in the *generating function* $G(x)$. E.g., if $G(x)$ is the generating function in (4.5) we have $[x^n]G(x) = a_n$ and

$$[x^n]G(x)\frac{1}{(1-x)} = \sum_{k=0}^{n} a_k. \tag{4.7}$$

For the rest of this text, it would be convenient to define two generating functions

$$F(x) = \sum_{k=1}^{\infty} f_k x^k \quad \text{and} \quad U(x) = \sum_{k=0}^{\infty} u_k x^k \tag{4.8}$$

which are generating the sequences $\{f_k\}$ and $\{u_k\}$. Note that $\{u_k\}$ is not a probability distribution, as $\sum u_k = \infty$. The generating functions of $\{f_k\}$ and $\{u_k\}$ are related by (Feller 1968b, pp. 311–312)

$$U(x) = \frac{1}{1-F(x)}. \tag{4.9}$$

By multiplying both sides of (4.4) by x^n, summing over the values of which the recurrence is valid (i.e., $n = r, r+1, r+2, \dots$), and recognizing $U(x)$ from (4.8), one gets

$$(U(x)-1)\,(1+px+p^2x^2+\cdots+p^{r-1}x^{r-1}) = p^r(x^r+x^{r+1}+\dots).$$

The two series, one on each side of the equation, are geometric, and one can see that

$$(U(x)-1)\cdot\frac{1-(px)^r}{1-px} = \frac{p^r x^r}{1-x}$$

or, by isolating $U(x)$ and introducing the variable $q = 1-p$, one may obtain

$$U(x) = \frac{1-x+qp^r x^{r+1}}{(1-x)(1-p^r x^r)}. \tag{4.10}$$

Finally, by substituting $U(x)$ in (4.9) and isolating $F(x)$, Feller (1968b, p. 323) presents the generating function

$$F(x) = \frac{p^r x^r(1-px)}{1-x+qp^r x^{r+1}} \tag{4.11}$$

which is a probability generating function, describing the probability for occurrences of *success runs* of length r in a sequence of *Bernoulli trials*.

Let $R_p(r,n)$ be the probability that a run of r consecutive successes appears in n independent trials. There is a beautiful formula given in terms of the coefficients of (4.11)

$$R_p(r,n) = \frac{[x^n]F(x)}{(1-x)} = \sum_{k=r}^{n} f_k \tag{4.12}$$

TABLE 4.1: The values generated by the function $2^\ell R_{1/2}(r,\ell)$ for $r = 0,\dots,\ell$ and $\ell = 1,2,\dots,12$.

	r												
ℓ	0	1	2	3	4	5	6	7	8	9	10	11	12
1	2	1	0	0	0	0	0	0	0	0	0	0	0
2	4	3	1	0	0	0	0	0	0	0	0	0	0
3	8	7	3	1	0	0	0	0	0	0	0	0	0
4	16	15	8	3	1	0	0	0	0	0	0	0	0
5	32	31	19	8	3	1	0	0	0	0	0	0	0
6	64	63	43	20	8	3	1	0	0	0	0	0	0
7	128	127	94	47	20	8	3	1	0	0	0	0	0
8	256	255	201	107	48	20	8	3	1	0	0	0	0
9	512	511	423	238	111	48	20	8	3	1	0	0	0
10	1 024	1 023	880	520	251	112	48	20	8	3	1	0	0
11	2 048	2 047	1 815	1 121	558	255	112	48	20	8	3	1	0
12	4 096	4 095	3 719	2 391	1 224	571	256	112	48	20	8	3	1

where one sums from $k = r$ instead of $k = 0$ as the r first coefficients are 0. The probability of a match between two randomly chosen bit strings of length ℓ, using the *r-contiguous match* rule, is given by setting $p = \frac{1}{2}$ and $n = \ell$.

Thus, if one selects a random bit string of a fixed length ℓ and performs a r-contiguous match, with a fixed r, against all possible strings of length ℓ (i.e., 2^ℓ different strings), the number of strings that will match is $2^\ell R_{1/2}(r,\ell)$. Table 4.1 shows the result of evaluating this formula for various values of ℓ and r.

Figure 4.4 on the facing page shows how the probability for a match between two randomly chosen strings consisting of symbols from an alphabet with cardinality $m = 2$, with varying string lengths ℓ and matching thresholds r using the *r-contiguous match* rule according to (4.12).

Not surprisingly, higher matching thresholds result in the creation of less general detectors. This fact was shown in (Forrest et al. 1994). An AIS which use a high matching threshold will thus need a larger number of detectors to effectively discriminate between *self* and *nonself*. On the other hand, such a system needs fewer retrials for each *mature detector* generated.

Figure 4.5 on the next page shows the difference between the probabilities for a match using the r-contiguous match rule according to (4.2) and (4.12).

4.4 NEGATIVE SELECTION

The *negative selection* mechanism in the IS is often used in AISs to perform *anomaly-based detection*. In (Forrest et al. 1994; Hofmeyr 1999; Hofmeyr and Forrest 2000) this is modeled by requiring that *valid detectors* are those *detectors* which does not detect self *agents* during *tolerization*. Figure 4.6 on the facing page shows this process. First, a detector is randomly generated, which means that its *receptors* may recognize anything. If the detector recognizes anything during tolerization, it dies. If the detector survives the tolerization period,

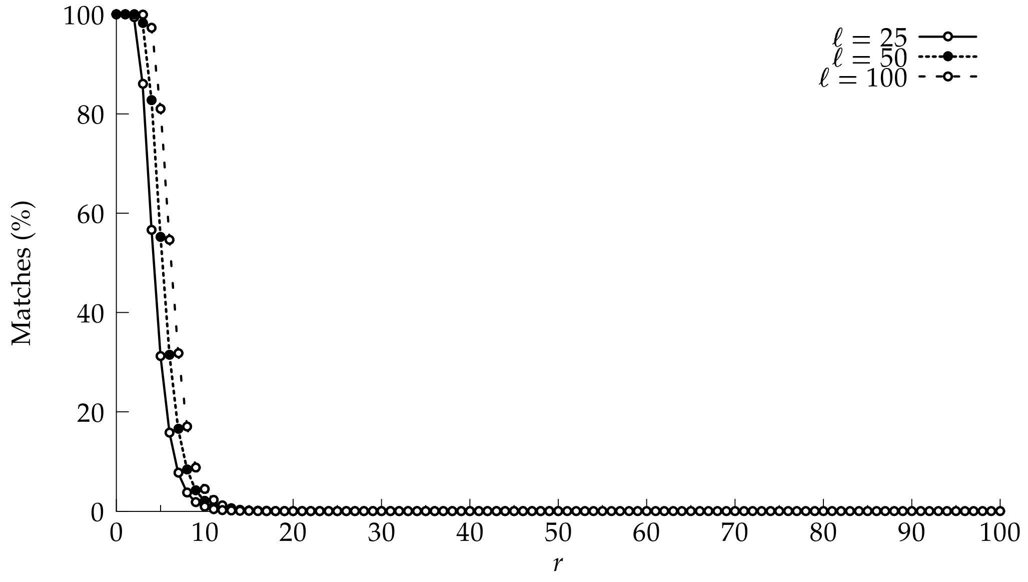


FIGURE 4.4: How the values of r and ℓ affects the probability of a match, using the r-contiguous match rule.

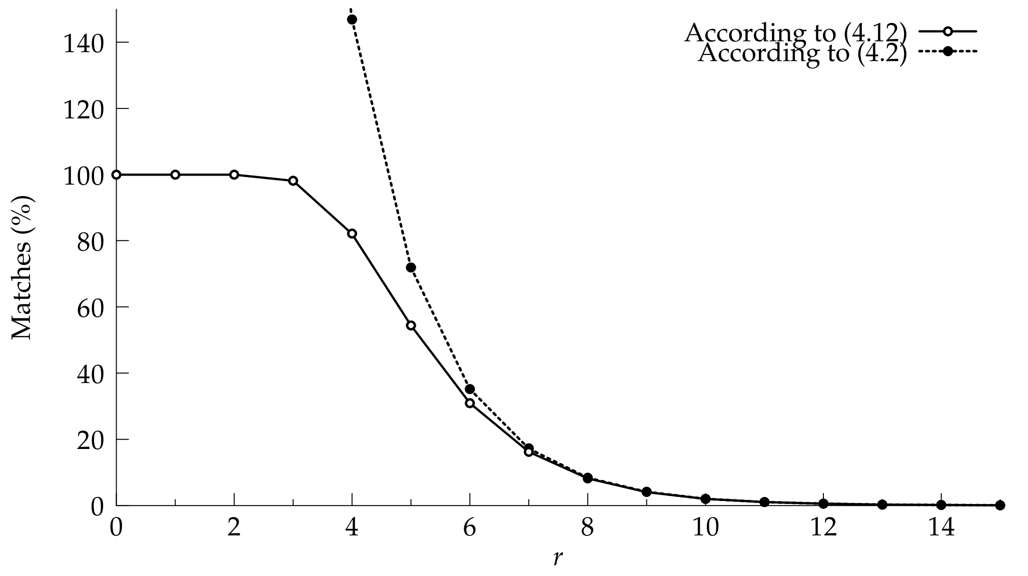


FIGURE 4.5: The difference between (4.2) and (4.12) in predicting the probability for a matching between two randomly chosen bit strings, using the r-contiguous match rule when r changes. In both cases $\ell = 49$.

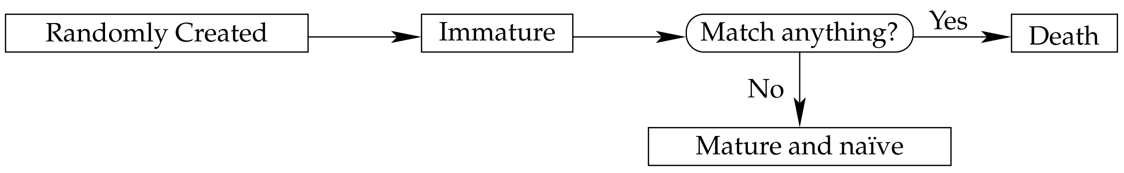


FIGURE 4.6: The *negative selection* process a *detector* undergo during *tolerization*.

it becomes a *mature* and naïve detector. It is called naïve because it has not detected any *pathogens* yet.

This use of *negative selection* is based on the assumption that if a *detector* recognizes anything during *tolerization*, what it matches is *self*. This way, the AIS implicitly *learns* that anything its *mature* and *memory detectors* match is *nonself*.

4.5 Clonal Selection and Somatic Hypermutation

As mentioned in Section 2.5, the combination of *clonal selection* and *somatic hypermutation* is an important factor in the process known as *affinity maturation*. The purpose of affinity maturation is to increase the *diversity* in the IS and the *affinity* between *detectors* and *agents* in the IS.

The utilization of clonal selection and somatic hypermutation to model affinity maturation in AISs applied to NID has been proposed, but not implemented, by Hofmeyr and Forrest (2000). Some experimental work exploring the role of somatic hypermutation in the IS has been performed though (Hightower, Forrest, and Perelson 1996).

4.5.1 Research Hypothesis

As explored above, with longer *T-cell* and Ab combining regions—i.e., higher r values—the AIS will be tuned towards higher specificity. And, vice versa, with shorter receptor recognition sites, the AIS will be tuned towards more general matching.

For example, with $\ell = 49$ and $r = 4$, the $R_{1/2}(r, \ell)$ evaluates to 0.566, while if $r = 15$ the same expression evaluates to 0.183×10^{-3}. As the *detectors* monitor several *peptides* over the course of time, this means that with low r-values, the detectors of the AIS will match virtually anything. On the other hand, with high values of r the detectors will match a much smaller set of *agents*.

Since the AIS's detector set is generated through *negative selection*, lower r-values result in a *higher* probability of matching *self* during *tolerization*, and higher r-values result in *lower* probability of matching self during tolerization. Thus, the lower the value of r, the more *retries* the AIS needs to generate each *mature detector*. The number random detectors that must be generated and tested are reported (Forrest and Hofmeyr 2001) to be "approximately exponential in the size of self." With higher r-values fewer detector generation retries are necessary, but a larger set of detectors are needed to achieve a certain level of *coverage* also increases.

This leads to a tradeoff situation, where lower r-values require a smaller detector set to achieve a certain coverage, while the AIS needs more retries for each valid detector it generates. Based on the role the combination of *clonal selection* and *somatic hypermutation* plays in the IS, it is assumed that if adopted to an AIS, the mechanisms will increase detector *diversity* and the *affinity* between detectors and agents. This leads to the central research question of this work:

Is it possible to achieve a higher level of coverage for a detector set generated with a fixed r-value by utilizing clonal selection and somatic hypermutation?

4.6 Other Approaches to Modeling Mechanisms in the Immune System

In Section 4.3, the use of bit strings for modeling *peptides* was introduced. Another approach, used by Kim and Bentley (2001), include the use of strings with $m = 10$ and $\ell = 33$. While the above discussion has been concerned with matching *peptides* at the *genotype* level, their AIS implementation performed matching on the *phenotype* level. Their work concluded that the *negative selection* algorithm was impractical for generating a sufficient number of valid *detectors* in a reasonable time. This result will be discussed in Section 7.5.

The role of the molecules of the MHC has been modeled by Hofmeyr and Forrest (2000) who let each location in their architecture for a distributed AIS filter each *agent's peptides* through a randomly generated *permutation masks* to generate alternative representations for each *peptide*. This mechanism was reported to introduce an important form of population-level *diversity*.

CHAPTER 5

IMPLEMENTATION OF AN ARTIFICIAL IMMUNE SYSTEM APPLIED TO NETWORK INTRUSION DETECTION

In this chapter, the author's implementation of DAIS—an AIS-based NID system—will be presented. DAIS was used to perform the experiments presented in Chapter 6. The source code for the program is included in Appendix A[1]. It should be noted that the program is licensed under the *GNU's Not Unix! (GNU) General Public License (GPL)* which is included in Appendix B.

Several aspects of DAIS are based on the proposed AIS architecture named *Artificial Immune System (ARTIS)*, presented by Hofmeyr and Forrest (2000). Other aspects, like *somatic hypermutation* and the notion of a *thymus* present in the AIS are adoptions, made by the author, based on the mechanisms of the IS presented in Chapter 2.

5.1 WHAT IS MONITORED?

As mentioned in Section 3.4, which protocols a NID system monitors, will greatly affect its abilities to detect intrusions. Another consideration is that if a NID system is to monitor everything, it will quickly use too many resources.

The DAIS adopts the *peptide* encoding used by (Hofmeyr 1999), which encodes *TCP SYN* (synchronize) packets. TCP SYN packets are sent to and from the participating hosts when a TCP connection is established. Figure 5.1 on the next page shows the process of establishing a TCP connection. First, the client (the initiating host) sends a SYN packet to the server (the host being contacted). Next, the server responds by sending a packet with both the SYN and the *ACK* (acknowledge) flag set to the client. Now the client knows that the server has received its initial SYN packet, and that it acknowledges the connection attempt. Finally, the client sends a packet with the ACK control flag set, and when the server receives this, a connection has been established (Postel 1981b). This procedure is called a *three-way handshake*.

The idea here is that a TCP connection cannot be established without the exchange of TCP SYN packets. Hence, the monitoring of SYN packets should provide a sensible way to give the AIS a sense of normal, or *self*, traffic and of abnormal, or *nonself*, network traffic.

[1]It will also be made available from the author's home page, `http://www.ranang.org/`.

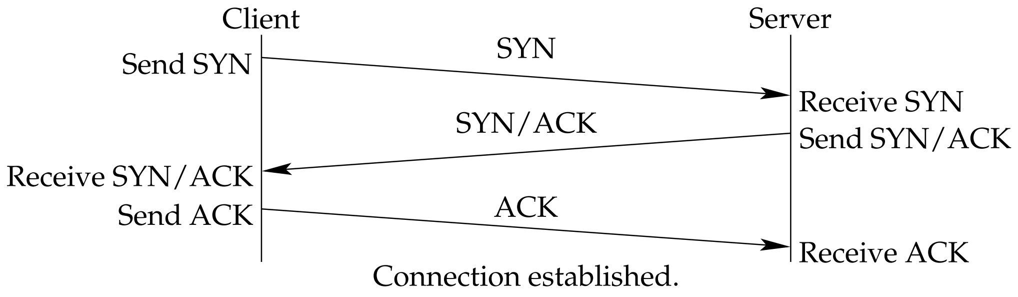


FIGURE 5.1: TCP three-way handshake. The time increases from the top to the bottom of the figure.

```
src > dst: flags data-seqno ack window urgent options
```

FIGURE 5.2: General format of a TCP line.

Since the data monitored by DAIS consists of SYN packets, *TCPdump* is used to obtain it. The *TCPdump* program outputs the headers of packets on a network interface. The general format of the TCP headers output by *TCPdump* is shown in Figure 5.2. The fields *src* and *dst* are the *source* and *destination* IP addresses and ports. The *flags* field, may be a combination of `S` (SYN), `F` (*FIN*), `P` (*PUSH*), `R` (*RST*) or a single '`.`' (no flags).

5.2 PEPTIDES

As mentioned in above, the encoding of SYN packets into an AIS peptide is adopted from (Hofmeyr 1999), where each connection is represented as a bit string, with $\ell = 49$. The schematic representation of a SYN connection is shown in Table 5.1. At least one of the computers participating in any of the connections has to be on the LAN; thus the first 8 bit of the peptide bit string represent an internal computer. These 8 bits represent the *least significant byte (LSB)* of that computer's *IP address*. This may be done under the assumption that all computers on the LAN share the same class C IP network address.

The next 32 bits are used to represent the other participating host, which may be either an external host or another internal host on the same LAN. If it is an *internal* host, only 8 bits are needed to represent it, but the whole address is used (32 bits).

If an external host *is* involved in the connection, it is always represented in the bits 8–39,

TABLE 5.1: *Peptide* representation of a *SYN*.

Bits	Description
0–7	Internal host or Internal host (server)
8–39	External host or Internal host (client)
40	Server flag
41–48	Service

Field	Possible Values
State	$\{\text{immature}, \text{mature}, \text{memory}, \text{active}, \text{dead}\}$
Peptide	ℓ random bits
Age	$[0, T_{\text{death}}]$
Activation level	α
Antigen	The agent a the detector binds to when activated

TABLE 5.2: Attributes of a detector.

thus bit 40, the next bit, is used to indicate whether or not the first computer is a server. If bit 40 is set, the first 0–7 bits represent an internal server, otherwise it is not set.

The final 8 bits, i.e., bits 41–48, represent the type of service the client is requesting. This field may have values in the range $[0, 255]$. On the *Internet* different hosts may run various services on their *IP ports*. These IP ports are usually running specific services, according to assignments of IP port numbers done by the *Internet Assigned Numbers Authority (IANA)*. IANA is an organization that maintains an online database[2] which defines a sequence of unique port assignments (Reynolds 2002). There are more services than can be represented by 8 bits, therefore the service field contains values who are mappings from IP ports numbers to a range of services actually occurring in the network traffic being monitored.

5.3 ARCHITECTURE OF DAIS

The architecture of DAIS is described in the following. It is similar to the *IS* in that it performs *anomaly-based detection*, consists of a multitude of detectors in a distributed environment, generates its set of *detectors* through *negative selection* and *clonal proliferation* combined with *somatic hypermutation* (or *affinity maturation*) and it is dependent on some kind of *costimulation* to increase its ability to discriminate between *self* and *nonself*.

However, it does not utilize *mobile detectors*, diversity among detectors through alternative representations (or permutations) and *location-based sensitivity*, as was proposed in (Hofmeyr and Forrest 2000). These concepts will be discussed later.

When DAIS is used to perform simulations, the distributed environment is modeled as a graph $G = (V, E)$, where each vertex $v \in V$ corresponds to a location $l \in L$. All communication between the nodes is done via the edges. Each location corresponds to a node, or host, in the network. If a *broadcast network* is simulated, the same traffic sent to and from a location l_i is sent over every edge $e \in E$ in the network; thus all other locations $\{l_n | \forall l_n \in L \wedge l_n \neq l_i\}$ will receive the same network traffic. If a *switched network* is simulated, the traffic travels along only the edges which are needed to establish a connection between the source and the destination host.

Each location consists of a population $D = D_{\text{I}} \cup D_{\text{N}} \cup D_{\text{M}}$, where D_{I} consists solely of *immature detectors*, D_{N} consists solely of *mature detector* and D_{M} consists solely of *memory detector*; thus $D_{\text{I}} \cap D_{\text{N}} \cap D_{\text{M}} = \emptyset$. The size of the sub-population $|D_{\text{I}}|$ is fixed at D_{Imax}, while $0 \leq |D_{\text{N}}| \leq D_{\text{Nmax}}$ and $0 \leq |D_{\text{M}}| \leq D_{\text{Mmax}}$. This way, D_{N} and D_{M} are dynamic populations, capable of expanding and shrinking over time.

[2]Available from `http://www.iana.org/`.

TABLE 5.3: Analogies between mechanisms in the IS and the implemented AIS.

Immune System	*Artificial Immune System*
Peptide, protein or epitope	Bit string
Receptor	Bit string
Binding/affinity	Approximate string matching
Somatic hypermutation	Somatic hypermutation
Memory cell	Memory detector
Signal 1	Activation threshold exceeded
Signal 2	Human operator
Pathogen detection	Detection event
Lymphocyte cloning	Detector replication
Lymphocyte mutation	Detector mutation
Affinity maturation	r-contiguous affinity maturation

When the populations D_{N} and D_{M} has reached their maximum size, and new individuals are to be inserted, a decision has to be made on what individual should be replaced by the new one. In the architecture described here, the insertions are performed by the *least recently used (LRU)* principle, described in (Knuth 1997b, p. 452). That is, the individual in the population that was used (e.g., activated) least recently is removed while the new individual takes up the newly freed space. This principle is also used by Hofmeyr and Forrest (2000) in ARTIS.

Each of the implemented *detectors* has a small set of attributes, as shown in Table 5.2 on the preceding page. The *state* of a detector may be either *immature*, *mature*, *memory* or *dead* and any of these may be combined with the state *active*. The *peptide* of a detector is the abstraction of the receptors on the surface of a *lymphocyte*. It is represented as a bit string of length 49. The *age* of a detector is measured in the number of *agents* monitored. Thus, a detector age of 25 000 corresponds approximately to 1 day in the real world. Every detector start their lives at age 0, and if they are not removed by any other cause, they die at age T_{death}. Another important attribute of the detectors is their activation level α. The reason it is used is that since matching in the AIS is approximate, there is an increased probability of matching against *self*. The activation level of a detector is increased every time it matches anything, and it is reduced gradually over time. The detector does not become activated until the activation level is above a certain threshold. This mechanism reduces the number of false positives. The last attribute of the detectors is the *antigen* which is a copy of the agent that caused the detector to become activated. This is necessary to perform the *somatic hypermutation* with *affinity maturation* as explained in Section 5.4.2. The analogies between mechanisms in DAIS and the IS is summarized in Table 5.3.

5.4 THE LIFE-CYCLE OF A DETECTOR

Figure 5.3 on the next page shows the life-cycle of a *detector*. First, a detector d is created with a randomly generated bit string. The state of d is then set to *immature*. It has now begun its *tolerization period*. If d matches *anything*, even just once, during this period, its state is changed to *dead*, just like *programmed cell death* (or *apoptosis*) in the biological

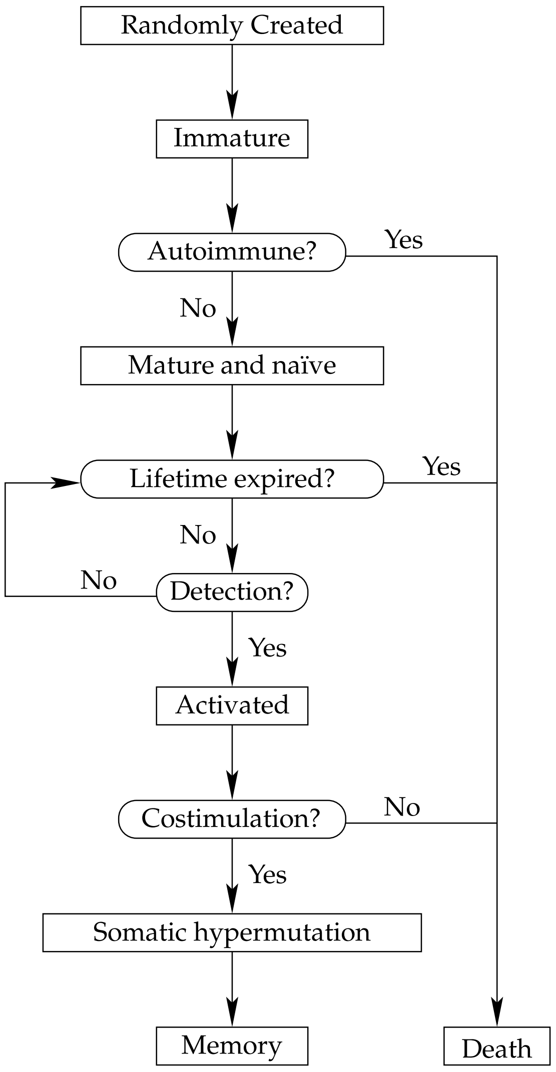


FIGURE 5.3: The *life-cycle* of a detector.

immune system. This is the implementation of *negative selection* in the AIS.

5.4.1 Costimulation

If d survives the *tolerization period*, its state is changed to *mature* and naïve. When a detector has reached the mature state, it is capable of detecting *nonself*. If the detector *matches* nonself frequently enough, its state is set to *active*. This corresponds to triggering an alarm if its activation level α reaches the activation threshold τ. Every time d matches an agent, α is increased by letting $\alpha \leftarrow \alpha \times \alpha^{+}$, where $\alpha^{+} \geq 1$. In a real-world application of this AIS-based NID, such an activation would result in an e-mail being sent to the administrator of the network. The operator is then supposed to provide the AIS with its *costimulation* (or "signal 2") within a given time γ, called the *costimulation delay* period. If the detector does not match the agent, its activation level is decreased as $\alpha \leftarrow \max(1, \alpha \times \alpha^{-})$, where $0 \leq \alpha^{-} \leq 1$. If d does not receive its costimulation signal, its state is set to *dead*. If, on the other hand, d receives costimulation, it enters the process of *clonal proliferation* and *somatic hypermutation*.

5.4.2 Somatic Hypermutation

During the *somatic hypermutation*, the detector undergoes clonal proliferation. The clonal proliferation is implemented simply by creating ξ *clones* of d. Each of the offspring undergoes a single point mutation with a probability of p_{mutate}. This mutation represents a *flip-mutation* variation operator as presented in (Michalewicz and Fogel 2000). To model the *affinity maturation*, only the offspring *detectors* with equal or higher affinity to the *agent* (that d recognized) than d is inserted into the population of *memory detectors* D_{M}; the others die. The *affinity* measure used is the longest *r-contiguous match* matching sequence between a detector and an agent. This means that every one of the detectors who survives this "battle" will match that agents on at least r contiguous locations.

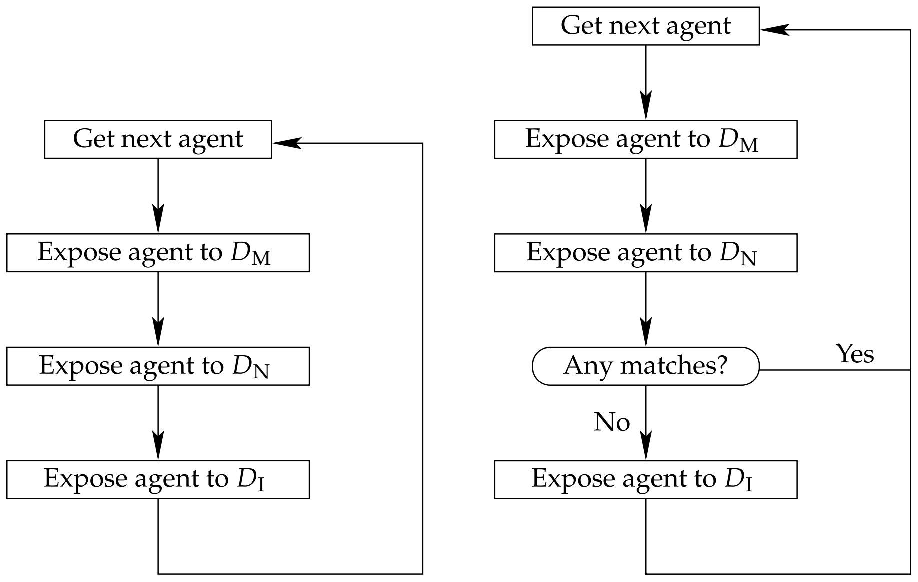
To implement the probability of an event occurring, a positive unsigned 32 bit integer `n` is assigned a random value in the range 0–2^{32}. This value is converted to a floating point value `p` according to the suggestions made by Press et al. (1992, p. 275, 276) as

$$p = \frac{n}{\texttt{RAND\_MAX} + 1.0}$$

where `RAND_MAX` is defined to be 2^{32}, the largest possible value returned from the random generator.

In the AIS, the *Mersenne Twister (MT)* algorithm, presented and discussed in (Matsumoto and Nishimura 1998), is used to generate uniformly distributed *pseudo-random numbers*. A uniform distribution on a finite set is a set where the occurrences of every element is equally likely. The implementation presented in (Matsumoto and Nishimura 1998)[3] has a period of $2^{19937} - 1$ and a 623-dimensional equidistribution property. Over the years, a lot of research dependent on random data generated by computers has suffered from poor *pseudo-random number generators*. What is meant by a *poor* pseudo-random number generator is a generator which produces sequences of pseudo-random numbers with short periods or with poor distribution; i.e., they are not even close to random. The period of a

[3]The AIS implementation described herein uses an updated version of the MT algorithm, made available from *Makoto Matsumoto*'s home page. The update was published 26 January 2002. It takes into consideration *Shawn J. Cokus*' optimizations, which makes the implementation four times faster than the ANSI-C `rand` function.


Get next agent
Expose agent to D_{M}
Expose agent to D_{N}
Expose agent to D_{I}
Get next agent
Expose agent to D_{M}
Expose agent to D_{N}
Any matches?
Yes
No
Expose agent to D_{I}

(a) *Without thymus.* (b) *With* thymus.

FIGURE 5.4: Model of thymus behavior in DAIS.

pseudo-random number generator is the number of numbers produced before the sequence of produced numbers is repeated.

As stated by Matsumoto and Nishimura (1998), "the initialization is care-free. This generator is as fast as other common generators, such as the standard ANSI-C `rand`, and it passed several statistic tests including *diehard*".

5.4.3 Thymus Behavior

In (Hofmeyr and Forrest 2000, p. 556) it is mentioned that one of the things not implemented in their AIS implementation, named LISYS, is the notion of a *thymus*. They argue that the (*self*) proteins present in the thymus is the analogy to a training set in the IS, and that the training of *lymphocytes* to distinguish between self and *nonself* will not work if nonself is frequently expressed in the thymus. They also write, when presenting how training of the detection system is performed in ARTIS (the architecture which LISYS is based upon), that during the *tolerization* period "the detector is exposed to the environment (self and possibly nonself strings), and if it matches any bit string it is eliminated."

The author has tried to implement the notion of a distributed thymus behavior. Figure 5.4 shows how the notion of a thymus has been modeled. It is based on the assumption that there is a lower frequency of nonself in the thymus than in the rest of the IS. Thus, the thymus in a way assures that *immature detectors* run a lower risk of being exposed to nonself proteins. Without this notion of thymus presence, immature detectors (part of D_{I}) are just as likely to be exposed to nonself as *mature* and *memory detectors*, as shown in Figure 5.4(a). However, by introducing a filter which prohibits the exposure of immature detectors to *agents* that have already been recognized as nonself by mature and memory detectors, as shown in Figure 5.4(b) on the preceding page, the immature detectors run a lower risk of being exposed to nonself.

It should be noted that the requirement for the "*thymus* filter" to filter out an *agent* is that one of the *mature* or *memory detectors* have *matched*, i.e., it is not necessary for an activation to occur.

5.5 The Speed of the r-Contiguous-Match Algorithm

The *receptors* of the *detectors* and the *agent peptides* in DAIS are implemented as arrays of data types with a length of 64 bits. The *r-contiguous match* algorithm is implemented using the `unsigned long long int` data type in the *C* programming language, which is 64 bits long on the *Intel x86* architecture.

```
r-CONTIGUOUS-MATCH($r, A, B$)
 1  if $r = 0$
 2    then return TRUE
 3  $\ell \leftarrow length[A]$
 4  $c \leftarrow 0$
 5  $X \leftarrow A \oplus B$
 6  for $i \leftarrow 0$ to $\ell$
 7  do if $X[i] = 1$
 8        then $c \leftarrow 0$
 9        else $c \leftarrow c + 1$
10             if $c = r$
11               then return TRUE
12  return FALSE
```

In the r-contiguous match algorithm shown above, A, B and X are bit strings of length ℓ, r is the match constraint and $\oplus$ denotes the *exclusive OR (XOR)* operator.

The XOR operator is implemented in hardware in the *Intel x86 family* of processors. This fact, combined with the use of the *left-shift* operator, makes the algorithm run very fast. This is important because the r-contiguous match function is called once for each detector which is to test for recognition of each and every agent (connection) the program monitors.

5.6 How to Use DAIS

It is possible to control all of the parameters mentioned above from the command line when starting DAIS. It has a self contained help-screen which is invoked by issuing the command:

```
./ais --help
```

The self-explanatory help screen is shown in Figure 5.5 on the next page. In addition to the parameter flags, DAIS expects a list of input files which are read in the order they are presented on the command line.

DAIS is able to read two different formats, namely *TCPdump* headers in *American standard code for information exchange (ASCII)* format and a special file format used for simulations, consisting of pre-formatted connection information as shown in Figure 5.6. The

```
Usage:  ./ais [ options ] [ inputfile ... ]
  -h, --help                 Display this usage information.
  -i, --info                 Only show the current configuration and exit.
  -t, --test                 Perform a match test.
  -r R, --r-value=R          Set the r of the r-contiguous-bits algorithm.
  -n N, --nodes=N            Set the number of nodes.
  -I N, --immature-detectors=N
                             Set the number of immature detectors per node.
  -N N, --mature-detectors=M  Set the maximum number of mature (naive) agents.
  -M N, --memory-detectors=M  Set the maximum number of mature agents.
  -l N, --detector-lifetime=N
                             Set the length of a detectors lifetime.
  -c N, --detector-childhood=N
                             Set the duration of a detectors childhood.
  -a R, --activation-inc=R   Set the factor for increasing the detector
                             activation.
  -d R, --activation-dec=R   Set the factor for decreasing the detector
                             activation.
  -D N, --costimulation-delay
                             Set the length of the period from activation
                             to costimulation is received.
  -f F, --format=F           Use TCP or AIS as file format.  Default is AIS.
  -p S, --node-ids=S         Read node IDs from file S.  (Switched LAN.)
  -y, --somatic-hypermutation
                             Perform somatic hypermutation (SH).
  -m, --sh-p-mutate=P        Let P be the probability of mutation under SH.
  -u, --sh-clones=N          Generate N clones under SH.
  -v, --sh-clones-select=N   Let N of the clones survive after SH.
  -C, --sh-competition       Simulate affinity competition under SH.
  -z, --thymus               Perform thymus simulation.
```

FIGURE 5.5: The help-screen of DAIS.

```
classification local_host:remote_host:first_is_server:service
```

FIGURE 5.6: The pre-formatted *peptide* file format.

`classification`-field may be either `S` (*self*) or `N` (*nonself*), while `local_host` is a number in the range $[0, 255]$ and `remote_host` is a IP version 4 *address*. The `first_is_server` and `service` fields are according to the *peptide encoding* described in Section 5.2.

5.6.1 Switched Ethernet versus Legacy Ethernet

Perhaps the most striking advancement in contemporary Ethernet networks is the use of *Switched Ethernets*. LISYS, the AIS-based NID system implemented by Hofmeyr (1999), does not support simulating *switched networks* environments. This support *is* implemented in DAIS, but for this feature to be truly useful, new *peptide encodings* are needed. This will be discussed in more detail in Chapter 7.

CHAPTER 6

RESULTS

This chapter describes the experiments performed in order to investigate the role of *somatic hypermutation* in the AIS and presents the results of these experiments. Some other IS-inspired mechanisms are also investigated, although they are not the main focus of this work.

When criticizing the most comprehensive evaluations of research on IDSs that has been performed to date, the 1998 and 1999 *Defense Advanced Research Projects Agency (DARPA)* Intrusion Detection System Evaluations, McHugh (2000) states that

> The operating points or curves obtained by plotting true positives against false positives is a relatively poor basis for characterizing research IDS systems since it provides no insight into the reasons for IDS performance (good or bad).

This is further emphasized by the fact that even if the measure is considered useful, there is no appropriate common denominator for both the true positive and false positive terms. A proposed approach is to require that both detections and non-detections of an IDS is reported. Where it is meaningful, this approach will be used in this chapter.

6.1 APPARATUS

The simulations were run on a cluster consisting of 1 master node and 43 computational nodes. Each simulation was done on a single node, but to be able to perform more simulations and thus cover a larger size of the parameter space, the cluster was used.

The nodes were connected to the master node through a dedicated *switched LAN*. The hosts' *hardware* were configured as shown in Table 6.1 on the following page. The *central processing units (CPUs)* were of type *Advanced Micro Devices (AMD) Athlon XP 1700+* and *Athlon MP 1600+* . The table shows the speed of the CPUs, the amount of *random access memory (RAM)* and *hard drive (HD)* space, and the speed of the NIC used for communication between the hosts. On 8 of the computational nodes the HDs utilize the *small computer system interface (SCSI)* for data transfers while the rest utilize the *integrated device electronics (IDE)* interface. All the hosts were running the *Source Mage GNU/Linux* OS, with the *Linux* kernel version 2.4.17.

TABLE 6.1: Hardware configuration of the hosts in the cluster used for running the simulations.

		CPU			HD		
Hosts	Type	Speed (GHz)	Model	RAM (GB)	Size (GB)	Interface	NIC (Mb)
1	Master	1 × 1.46	XP 1700+	2	3 × 80	IDE	1 000
8	Node	1 × 1.40	MP 1600+	1	1 × 18	SCSI	100
6	Node	2 × 1.40	MP 1600+	1	3 × 18	SCSI	100
1	Node	2 × 1.40	MP 1600+	1	2 × 18	SCSI	100
12	Node	1 × 1.46	XP 1700+	1	1 × 40	IDE	100
16	Node	1 × 1.46	XP 1700+	2	1 × 40	IDE	100
44	—	73.14	—	61	1 864	—	5 300

TABLE 6.2: Self data sets used in the simulations.

Set Type	Symbol	All Strings	Unique Strings	Not in Train
Train	S_{Train}	1 266 000	3 763	—
Test	S_{Test}	182 629	626	137

6.2 THE EXPERIMENTAL DATA SETS

The *data sets* used in the experiments was generously provided by *Stephanie Forrest* and *Justin Balthrop* (and others), at University of New Mexico (UNM). The same data sets were used in (Hofmeyr and Forrest 2000) and (Hofmeyr 1999). The sets are divided into two groups, one where each set consists of self data only and one where the sets consist of both self and nonself data. An overview of the self and the nonself data sets are given in Table 6.2 and Table 6.3 on the facing page respectively.

The set S_{Train} was collected over a period of 50 days. The original self set was much larger, but some traffic, caused by noisy traffic sources like *Hypertext Transfer Protocol (HTTP)* and *File Transfer Protocol (FTP)* servers, were filtered out. These kind of servers are continually communicating with new hosts; thus there is no stable definition of what *normal* network traffic is, which is one of the prerequisite conditions for the AIS to succeed. Note that only HTTP and FTP traffic to and from those servers were filtered out; the other ports were still monitored. It should also be noted that such traffic to and from hosts on the LAN, which was *not* supposed to run those kind of services, was *not* filtered out from the *data sets*.

Because the connections in training set S_{Train} was gathered in 50 days, it is assumed in the following that the average number of connections per day is $|S_{\text{Train}}|/50 \approx 25\,000$.

6.3 THE EXPERIMENTS

Two experiments were performed. The first experiment consisted of 2 640 simulations while the second experiment consisted of 3 360 simulations. The total size of all the log files gen-

TABLE 6.3: Nonself data sets used in the simulations.

Intrusion Type	Symbol	Strings	Fraction Nonself	Fraction Unique Nonself
Address probing	N_{AP}	8 600	0.540	0.340
Limited probing 1	N_{LP_1}	1 174	0.617	0.118
Limited probing 2	N_{LP_2}	114	1.000	0.842
Limited probing 3	N_{LP_3}	1 317	0.101	0.002
Port scanning	N_{PS}	2 966	0.435	0.196
Single port probing 1	N_{SP_1}	36	1.000	0.833
Single port probing 2	N_{SP_2}	285	0.165	0.130

erated during these experiments was approximately 6.30 *terabyte (TB)*[1]. On the average the size of the log file generated by each simulation was approximately 1.08 *gigabyte (GB)*.

A few measures of goodness will be introduced and used in the following. For each result set, two measures of goodness per data set were calculated; these measures are the rates of false and true alarms. During these experiments, *one* alarm (true of false) was counted if *one or more detectors* were activated at *one or more* locations of the AIS while the system was presented to one *agent*. Hence, a true or false *alarm* is counted *once* per connection (or time unit) for the whole AIS seen as a whole.

The rate of false alarms $A^-_{\delta_n}$ for data set δ_n is defined as

$$A^-_{\delta_n} = \frac{alarm^-_{\delta_n}}{|\{a\colon a \in \delta_n \cap U_S\}|} \tag{6.1}$$

where $alarm^-_{\delta_n}$ is the number of false alarms generated while monitoring the data set $\delta_n \in \Delta$ and $|\{a\colon a \in \delta_n \cap U_S\}|$ is the number of *self* agents in δ_n and hence the highest number of false alarms. Likewise, the rate of true alarms $A^+_{\delta_n}$ is defined as

$$A^+_{\delta_n} = \frac{alarm^+_{\delta_n}}{|\{a\colon a \in \delta_n \cap U_N\}|} \tag{6.2}$$

where $alarm^+_{\delta_n}$ is the number of true alarms generated while monitoring the data set $\delta_n \in \Delta$ and $|\{a\colon a \in \delta_n \cap U_N\}|$ is the number of *nonself* agents in δ_n.

For each data set δ_n, the utility functions $\Delta A^+_{\delta_n}$ and $\Delta A^-_{\delta_n}$ are calculated. These are the differences in true and false alarm rates, respectively, when *somatic hypermutation* is used and is not used. They are defined as

$$\Delta A^-_{\delta_n} = A^-_{\delta_n,\mathrm{plain}} - A^-_{\delta_n,\mathrm{SH}} \tag{6.3}$$

and

$$\Delta A^+_{\delta_n} = A^+_{\delta_n,\mathrm{SH}} - A^+_{\delta_n,\mathrm{plain}} \tag{6.4}$$

where *SH* and *plain* indicates if the alarm rate was generated using or not using somatic hypermutation. This means, that if the utilization of somatic hypermutation is successful, both ΔA^+ and ΔA^- should be positive.

[1]The total HD space actually used was only 60.5 *GB* due to compression with the *gzip* utility.

The probability p_{mutate} of a single point mutation in a bit string, during the clonal proliferation caused by somatic hypermutation, is set to 1.0 in all the experiments. This was done due to the small scale of the AIS compared to the scale of the IS. All the *peptides* used in the experiments are representations of *TCP SYN* packets encoded as bit strings of length $\ell = 49$ unless stated otherwise.

It should also be noted that in the results presented for each simulation, the events happening while DAIS processes S_{Train} (for the first time) are ignored.

6.4 The Effect of Somatic Hypermutation on Alarm Rates

To see how *somatic hypermutation* affects the number of true and false alarms; i.e., the effects at the level that a user of DAIS would interface, an experiment was set up as described below. An alarm occurs when the activation level α of one of the *detectors* reach the activation threshold τ; i.e., when $\alpha \geq \tau$.

The simulated NID was distributed over $|L| = 25$ locations. Each $l \in L$ had a population $D = D_{\text{I}} \cup D_{\text{N}} \cup D_{\text{M}}$ of detectors, where the maximum number of *immature detectors* $D_{\text{Imax}} = 40$, the maximum number of *mature detectors* $D_{\text{Nmax}} = 70$ and the maximum number of *memory detectors* $D_{\text{Mmax}} = 100$.

The experiment used the data sets S_{Train}, N_{AP}, N_{PS}, S_{Train}, N_{PS} and N_{AP} as input (in that order). This sequence, with repetitive exposure to *nonself*, was chosen to see if there were any signs of a *secondary response* like one can see in the IS. The total duration of the simulated period was 2 555 132 time steps, or a little over 102 days.

The matching constraint r varied in the range $[4, 15]$. The *tolerization period* T was set to the values in the set $\{10\,000, 25\,000, 50\,000\}$. Thus, the values of T corresponds to approximately 9.6 hours, 1 and 2 days accordingly. The lifetime T_{death} of the detectors was set to 5 and 10 times T. The various T and T_{death} combinations used during the experiment are shown in Table 6.4 on the next page.

The number of clones ξ that was generated during *affinity maturation*, when utilized, was set to 5 and 10. The *costimulation delay* γ was set to 0 and 25 000, which corresponds to receiving the *costimulation* signal immediately and after 1 day, respectively. Two different (α^+, α^-) combinations were used too. In the first combination $\alpha^+ = 1.2$ and $\alpha^- = 0.8$ and in the second one $\alpha^+ = 1.333$ and $\alpha^- = 0.9$. In the following, such a tuple will be referred to as a *sensitivity tuple*. The first sensitivity tuple has a slower activation rate and a faster decline, while the second sensitivity tuple simulates the opposite behavior.

All of the above parameters were organized so that every parameter was tested against all possible values of the other parameters. Finally, every simulation was performed four times; *With* and *without* the modeled notion of a *thymus*, and both of these combinations was tested once *with* somatic hypermutation and once *without*. This resulted in 2 640 unique simulations.

6.4.1 Results

To get an overview of the effect of the *somatic hypermutation*, the utility functions $\Delta A^-_{\delta_n}$ and $\Delta A^+_{\delta_n}$ were averaged over all the n data sets in each simulation. Hence, the values $\overline{\Delta A^-}$ and

Position	T	T_{death}
0	10 000	50 000
1	10 000	100 000
2	25 000	125 000
3	25 000	250 000
4	50 000	250 000
5	50 000	500 000

TABLE 6.4: The combinations of T and T_{death} used in the simulations and their ordering in the overview graphs.

$\overline{\Delta A^+}$ are defined as

$$\overline{\Delta A^-} = \frac{\Delta A^-_{\delta_1} + \Delta A^-_{\delta_2} + \cdots + \Delta A^-_{\delta_n}}{n} \tag{6.5}$$

and

$$\overline{\Delta A^+} = \frac{\Delta A^+_{\delta_1} + \Delta A^+_{\delta_2} + \cdots + \Delta A^+_{\delta_n}}{n} \tag{6.6}$$

respectively.

Figures 6.1 and 6.2 on the following page show $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$ for simulations where the *costimulation delay* $\gamma = 0$ and the number of clones generated during *clonal proliferation* $\xi = 10$. In Figure 6.1 the *sensitivity tuple* $(\alpha^+, \alpha^-) = (1.2, 0.8)$, while in Figure 6.2 $(\alpha^+, \alpha^-) = (1.333, 0.9)$. Note that because of the definition of $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$, the higher the value of a data point, the better the effect of *somatic hypermutation*.

It should be noted that in the graphs summarizing $\overline{\Delta A^\pm}$ values like this, the values (parameters) along the horizontal axis are arranged by increasing values of r, but for each tick indicating a new r-value, there are 6 data points, arranged in the order shown in Table 6.4. Thus, for all the figures of this kind for this experiment, the values of T and T_{death} are ordered as shown in the table.

Because of the huge amount of results generated by this simulation, the trends described in the following are based on the set of all the results as a whole, while the graphs presented and commented herein should be representative for these trends.

The graphs in figures 6.1 and 6.2 describe the general trend in the effect of utilizing somatic hypermutation when *no* thymus behavior is used, but affinity maturation *is* utilized. First of all, the use of somatic hypermutation has a distinctive negative effect on $\overline{\Delta A^+}$ for values of $r \in [6, 8]$. It should be noted, though, that the negative effect of using somatic hypermutation in this region seems to decrease with longer *tolerization periods* T and lifetimes T_{death} for the *detectors*. It also seems that the negative effect in $\overline{\Delta A^+}$ in this region is slightly biased towards higher values of r when using the $(\alpha^+, \alpha^-) = (1.333, 0.9)$ sensitivity tuple. With respect to $\overline{\Delta A^-}$, somatic hypermutation seems to have a negative effect for all values of r in the range $[6, 14]$, and that the negative effect is most significant in the range $r \in [8, 13]$. The most couraging trend in the graph is however the effect of somatic hypermutation on $\overline{\Delta A^+}$ where $r \geq 10$, independently of the sensitivity tuple used. This effect seems to increase significantly for values of $r \geq 13$.

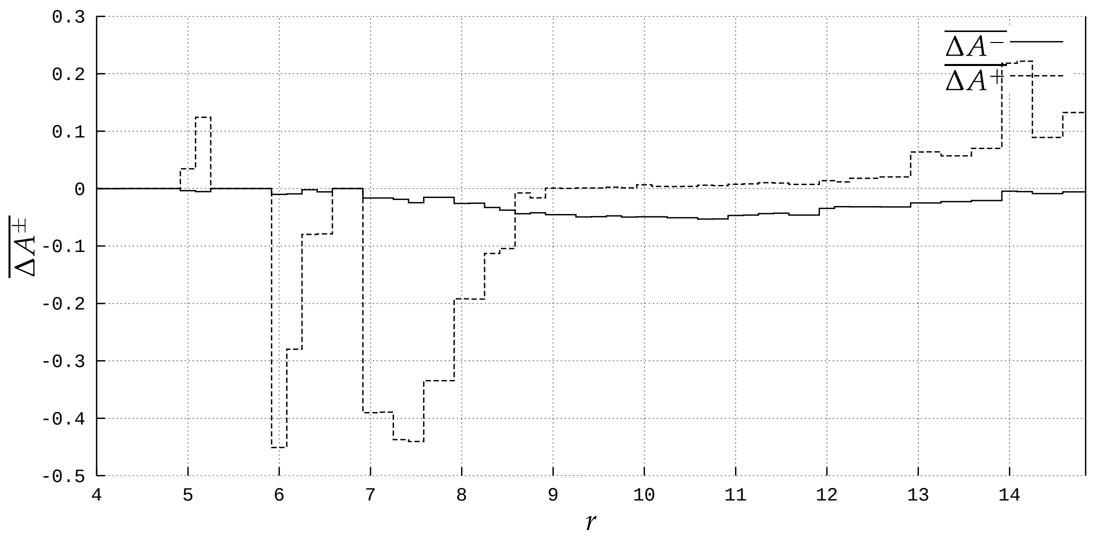


FIGURE 6.1: How $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^{+}, \alpha^{-}) = (1.2, 0.8)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on the page before. The higher the value of both $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$, the higher the performance gain from using somatic hypermutation. No thymus behavior was utilized, but affinity maturation was used.

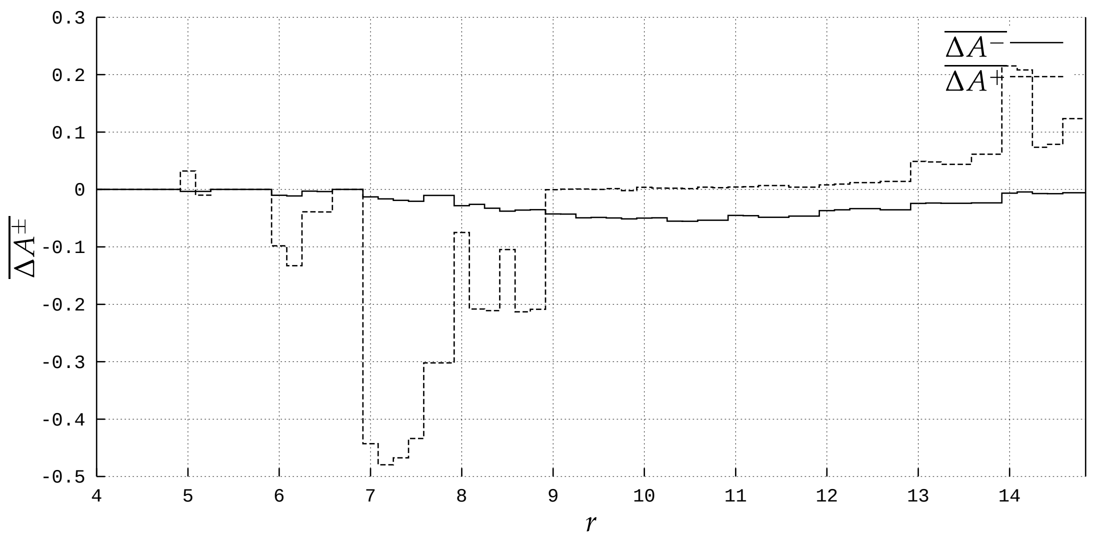


FIGURE 6.2: How $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^{+}, \alpha^{-}) = (1.333, 0.9)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on the page before. The higher the value of both $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$, the higher the performance gain from using somatic hypermutation. No thymus behavior was utilized, but affinity maturation was used.

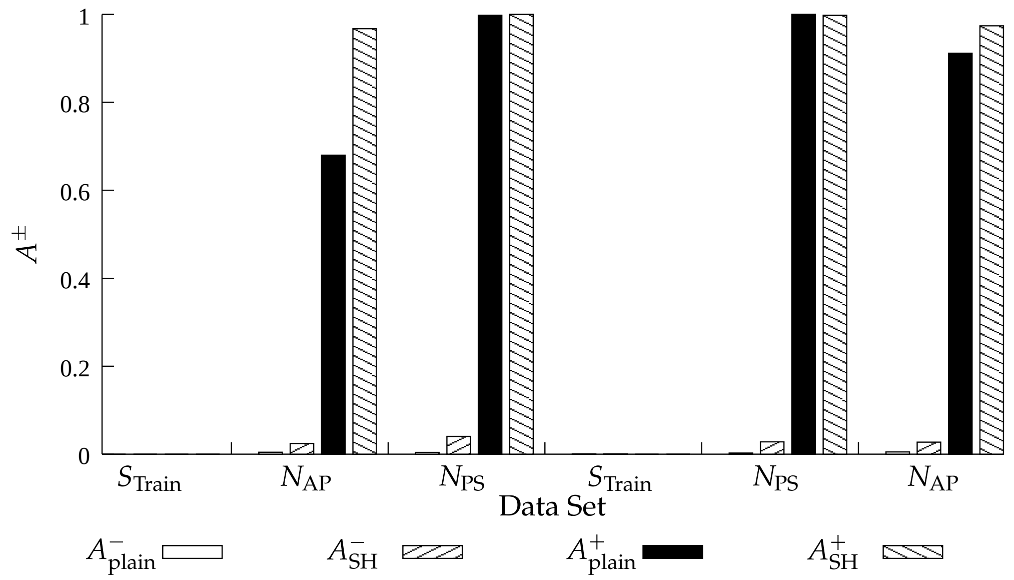


FIGURE 6.3: How A^-_{plain}, A^-_{SH}, A^+_{plain} and A^+_{SH} for each data set vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were the same as the ones used in Figure 6.1 on the preceding page, with the combination of $r = 13$, $T = 50\,000$ and $T_{\text{death}} = 250\,000$.

Figure 6.3 shows how the alarm rates A^-_{plain}, A^-_{SH}, A^+_{plain} and A^+_{SH} vary when the different data sets of the simulation are monitored. The parameters used were the same as in Figure 6.1 on the preceding page, except that this figure shows the details behind one individual data point ($r = 13$, $T = 50\,000$ and $T_{\text{death}} = 250\,000$). The thickness of the bars in the bar chart does not reflect the number of *agents* in each *data set*, and the behavior of the AIS during the first occurrence of S_{Train} is ignored, since the traffic monitored during this period was used as the training set of the AIS. The figure shows that when using somatic hypermutation the simulation has a significantly higher number of true alarms, but also a slightly higher number of false alarms. There are also signs of something which resembles a *secondary response* in the true alarm rates; the number of A^+ is higher during the second exposure to the agents in the data set N_{AP} than during the first response (note that the data sets N_{AP} and N_{PS} are encountered in reverse order the second time). It should also be noted that there is an insignificant number of false alarms during the second encounter of exposure to the S_{Train} data set.

The graph in Figure 6.4 on the following page shows how $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 25\,000$, $(\alpha^+, \alpha^-) = (1.2, 0.8)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. One of the most striking features of this graph is the values on the $\overline{\Delta A^{\pm}}$ axis. These values are extremely small, compared to the previous examples. The second thing to notice is that the clean pattern shown in the previous graphs is not present. These features are characteristic for all the results where the *costimulation delay* $\gamma = 25\,000$, i.e., approximately 1 day. There is one trend that seems to be persistent in these graphs where long costimulation delay were used, and that is the tendency that the absolute values of $\overline{\Delta A^{\pm}}$ get higher, the higher the value of r.

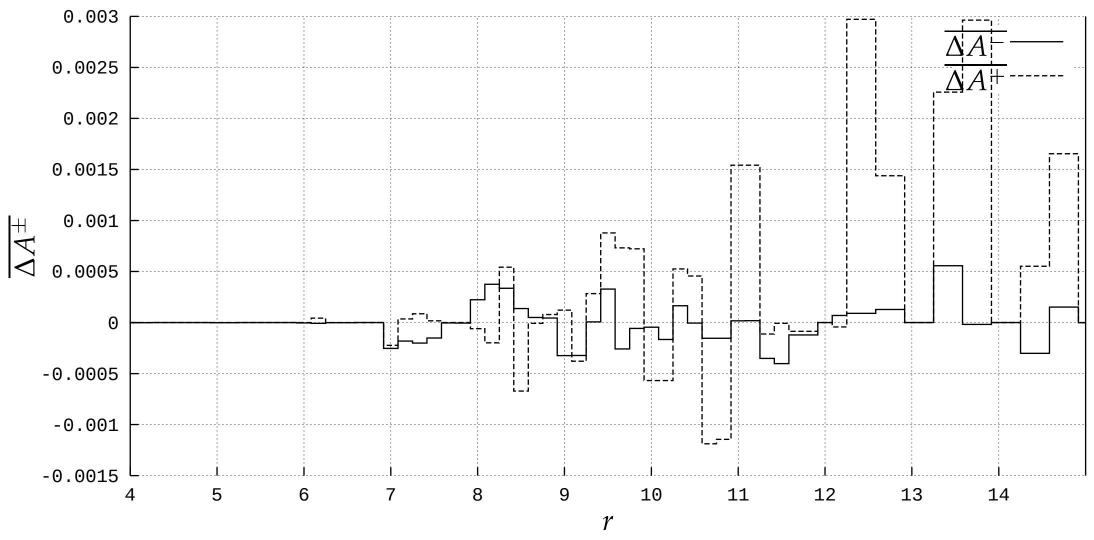


FIGURE 6.4: How $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 25\,000$, $(\alpha^{+}, \alpha^{-}) = (1.2, 0.8)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The higher the value of both $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$, the higher the performance gain from using somatic hypermutation. No thymus behavior was utilized, but affinity maturation was used.

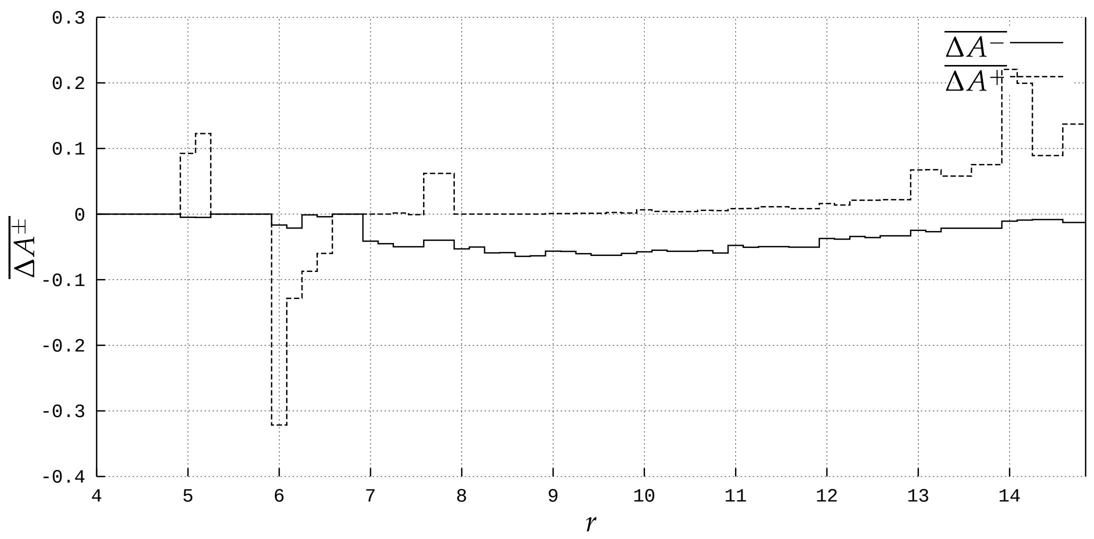


FIGURE 6.5: How $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^{+}, \alpha^{-}) = (1.2, 0.8)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The higher the value of both $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$, the higher the performance gain from using somatic hypermutation. No thymus behavior was utilized, and no affinity maturation was used.

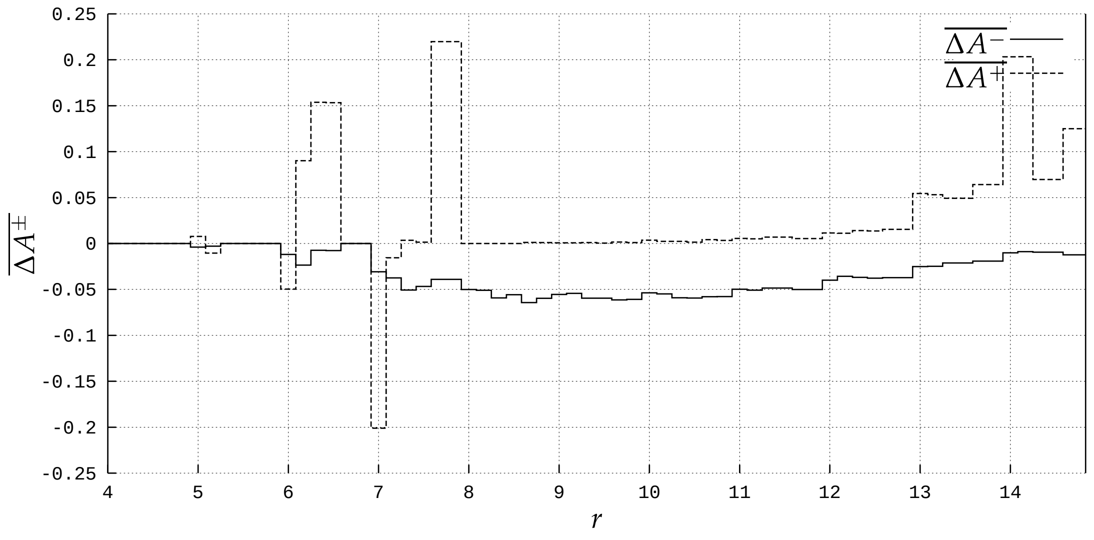


FIGURE 6.6: How $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^+, \alpha^-) = (1.333, 0.9)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The higher the value of both $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$, the higher the performance gain from using somatic hypermutation. No thymus behavior was utilized, and no affinity maturation was used.

The experiment also explored an alternative implementation of somatic hypermutation that does not use affinity maturation. During *clonal proliferation all detector* offspring became *memory detectors* without the requirement that they expressed equal or higher *affinity* to the triggering *agent* than the detector first recognizing the agent. The results of these simulations are shown in figures 6.5 and 6.6. Surprisingly, this approach did not affect the $\overline{\Delta A^-}$ values as much as expected. But, as is shown in Figure 6.6, when a more aggressive *sensitivity tuple* $(1.333, 0.9)$ is used, some of the positive characteristics become more apparent (compare the range $r \in [6, 8]$ in the two figures). Note that the $\overline{\Delta A^\pm}$ axis is different in the two figures. In addition, it should be noted that the negative effect seen in $\overline{A^+}$ in figures 6.1 and 6.2 on page 42 when $r \in [6, 8]$ seems significantly reduced when no affinity maturation was utilized.

Finally, the role of the modeled thymus behavior was tested. Figure 6.7 on the next page shows the trends when this feature was used. There were few dramatic changes in the behavior of the AIS by utilizing this notion of a thymus, but the gain in $\overline{\Delta A^+}$ that somatic hypermutation gave in Figure 6.1 on page 42 for $r = 14$ seems to be slightly amplified in these simulations.

To get an impression of the overall behavior of the AIS during these simulations, two additional measures are introduced. The average alarm rates $\overline{A^-}$ and $\overline{A^+}$ are defined as

$$\overline{A^-} = \frac{A^-_{\delta_1} + A^-_{\delta_2} + \cdots + A^-_{\delta_n}}{n} \tag{6.7}$$

and

$$\overline{A^+} = \frac{A^+_{\delta_1} + A^+_{\delta_2} + \cdots + A^+_{\delta_n}}{n} \tag{6.8}$$

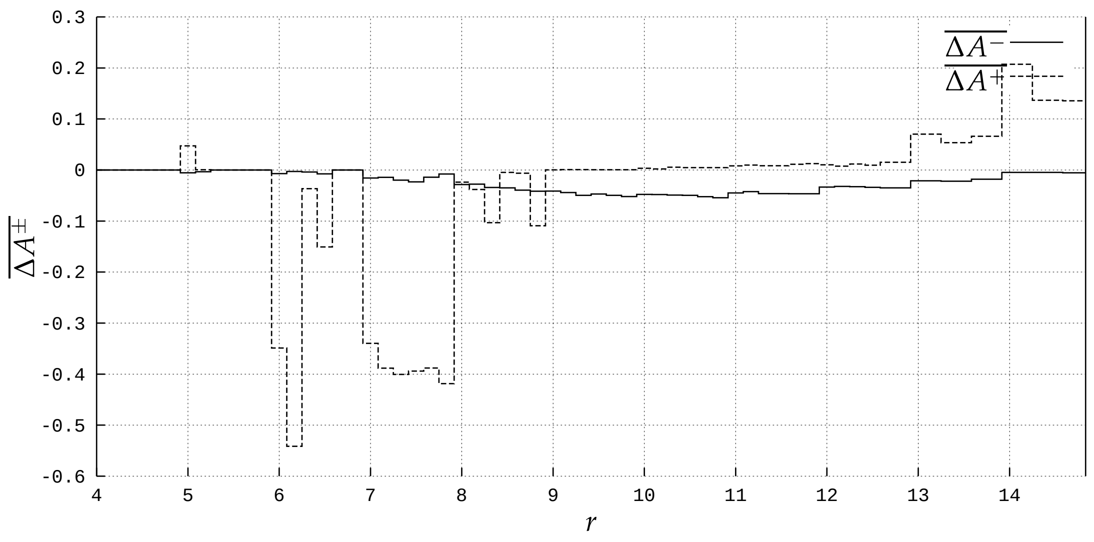


FIGURE 6.7: How $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^+, \alpha^-) = (1.2, 0.8)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The higher the value of both $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$, the higher the performance gain from using somatic hypermutation. Both thymus behavior and affinity maturation were utilized.

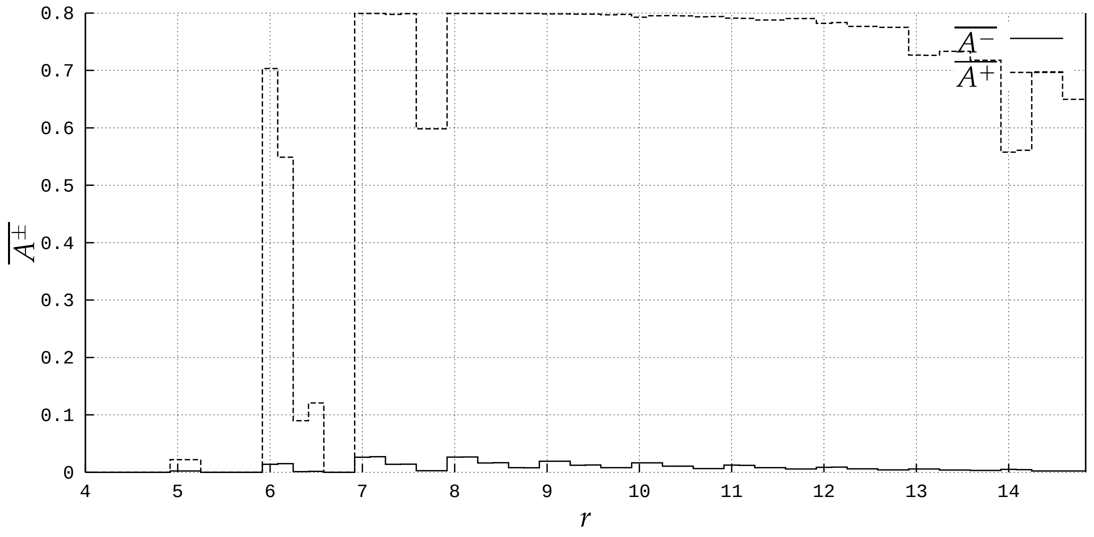


FIGURE 6.8: How $\overline{A^+}$ and $\overline{A^-}$ vary when no somatic hypermutation is used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^+, \alpha^-) = (1.2, 0.8)$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The same parameters were used in Figure 6.1 on page 42. No thymus behavior was utilized.

respectively. Figure 6.8 on the preceding page shows how $\overline{A^+}$ and $\overline{A^-}$ vary when *no* somatic hypermutation is used. The parameters used for that set of simulations were $\gamma = 0$, $(\alpha^+, \alpha^-) = (1.2, 0.8)$, and no thymus behavior was utilized. The other values, T and T_{death}, were ordered as shown in Table 6.4 on page 41. One of the most significant trends to notice is that there are close to zero true alarms $\overline{A^+}$ when $r < 6$, and that for $r = 6$ the AIS is quite effective for *detectors* with short *tolerization* periods T and short lifetimes T_{death}. The decline in $\overline{A^+}$, with longer tolerization periods and lifetimes that was seen when $r = 6$, is also seen for $r = 7$, although the general level of $\overline{A^+}$ is higher for this value of r. When $8 \leq r \leq 10$ the level of $\overline{A^+}$ is at its maximum, because

$$\overline{A^+} = \frac{A^+_{N_{\text{AP}}} + A^+_{N_{\text{PS}}} + A^+_{S_{\text{Train}}} + A^+_{N_{\text{PS}}} + A^+_{N_{\text{AP}}}}{5} \quad \text{and} \quad A^+_{S_{\text{Train}}} = 0.$$

This means that when all the alarm rates for the *nonself* data sets equals 1.0, the maximum level of $\overline{A^+} = 0.8$. Where $r \geq 11$ the trend is that there are fewer true alarms the higher the value of r. With respect to the false alarm rates, these seem to decline with the length of both the *tolerization* period and the lifetime of the *detectors*, with maximum level of false alarms where $r \in [7, 8]$. Where $r \geq 13$, there are close to zero false positives.

6.5 Different Network Traffic and Higher Values of r

Another experiment, consisting of 3 360 simulations, was conducted. Here, an AIS was simulated with different data sets as input than in the experiment described in Section 6.4. Now the input was S_{Train}, N_{LP_1}, S_{Test}, N_{LP_2}, N_{AP}, N_{PS}, S_{Test}, N_{LP_3}, N_{AP}, N_{LP_3}, N_{PS} and N_{SP_1} (in that order). Thus, this simulated network traffic contained more varied *nonself* traffic than the first experiment. In addition, because the S_{Test} data set was used, this experiment would test how the AIS behaved when parts of the *self* set was not present in the training set. The total duration of the simulated situation was 1 658 348 time steps, which corresponds to approximately 66.3 days. This means that these simulations were much shorter than the ones monitored during the first experiment.

Most of the parameters were kept identical to the ones in the first experiment, but some of them where changed. Notably, the matching constraint $r \in [4, 17]$, the *sensitivity tuple*s were $(1.2, 0.8)$ (not changed) and $(1.333, 0.8)$, and the maximum number of *immature detectors* $D_{\text{Imax}} = 30$. The other parameters were kept. The different combinations of *tolerization* periods T and the *detector lifetimes* T_{death} was still arranged as shown in Table 6.4 on page 41. The motivation behind this experiment was to see how the AIS behaved with different and more varied input, and if the positive trend seen in the first experiment, with high values of $\overline{\Delta A^+}$ when using *somatic hypermutation* in the range $r \geq 10$, would improve with even higher values of r.

6.5.1 Results

In general, the results of the simulations performed during this experiment strengthened the findings of the first experiment. In general the trends seen in the results from the first experiment for $r \in [4, 14]$ are very much the same as the ones seen in the results from the second experiment. The results of this experiment are presented in figures 6.9 to 6.20 on pages 48–54.

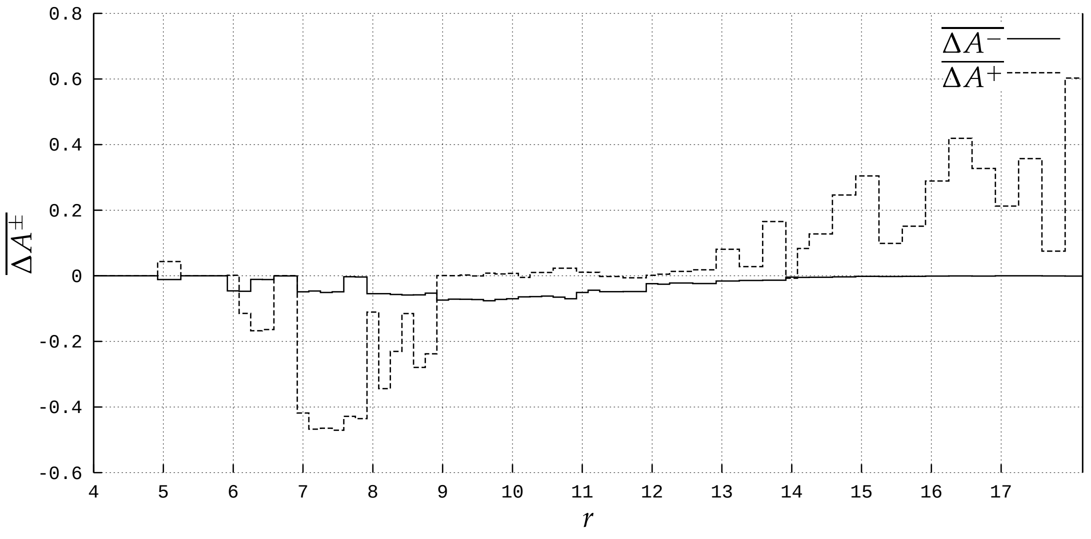


FIGURE 6.9: How $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^+, \alpha^-) = (1.2, 0.8)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The higher the value of both $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$, the higher the performance gain from using somatic hypermutation. No thymus behavior was utilized, but affinity maturation was used.

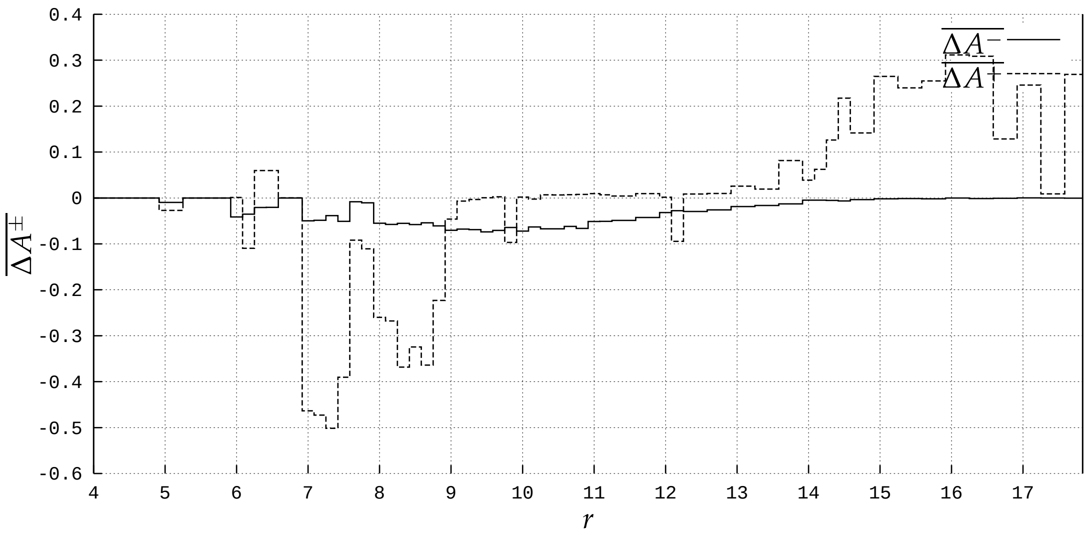


FIGURE 6.10: How $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^+, \alpha^-) = (1.333, 0.8)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The higher the value of both $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$, the higher the performance gain from using somatic hypermutation. No thymus behavior was utilized, but affinity maturation was used.

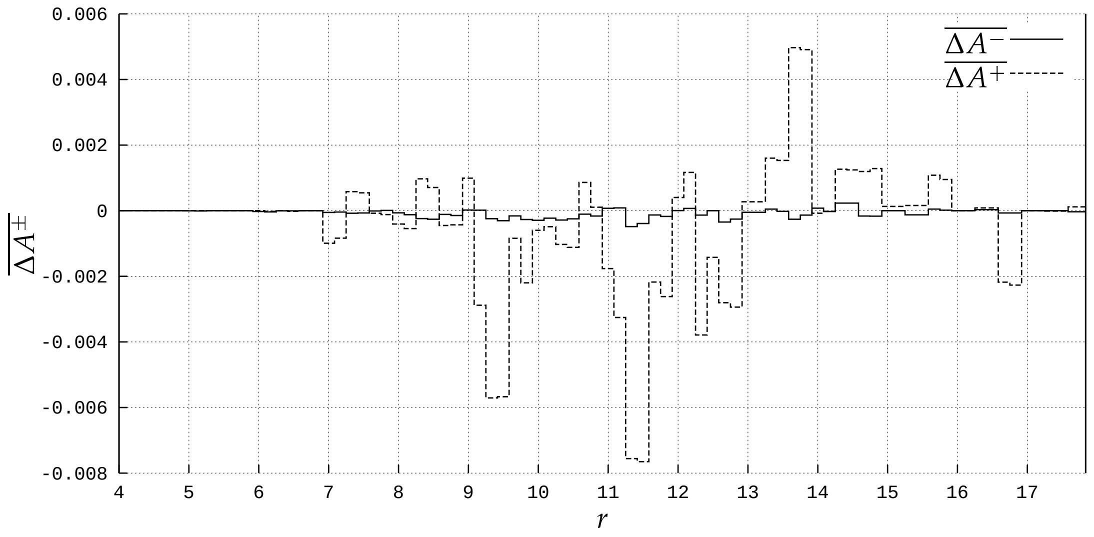


FIGURE 6.11: How $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 25\,000$, $(\alpha^+, \alpha^-) = (1.2, 0.8)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The higher the value of both $\overline{\Delta A^+}$ and $\overline{\Delta A^-}$, the higher the performance gain from using somatic hypermutation. No thymus behavior was utilized, but affinity maturation was used.

When comparing Figure 6.1 on page 42 to Figure 6.9 on the preceding page, which share the same parameter settings, one can see that in the region $r \in [4, 14]$ the graphs are very similar, and that the positive trends seen where $r \geq 13$ in the first graph continues even further and more positive in the second graph.

The new *sensitivity tuple* $(1.333, 0.8)$ seems to have the same effect on the number of both true and false alarms in Figure 6.10 compared to Figure 6.9 as the sensitivity tuple $(1.333, 0.9)$ had in Figure 6.2 compared to Figure 6.1 on page 42.

Just as seen in the first experiment, when the *costimulation delay* $\gamma = 25\,000$ the signature of the simulations become less regular. There is also a tendency that when the sensitivity tuple is more aggressive; i.e., $(\alpha^+, \alpha^-) = (1.333, 0.8)$ there are higher true alarm rates than when it is not. It is also a tendency that the false alarm rate is little affected by the use of somatic hypermutation when $\gamma = 25\,000$. These trends may be seen in Figure 6.11 and in Figure 6.12 on the following page. The values on the $\overline{\Delta A^\pm}$ axes are much smaller than the ones seen when $\gamma = 0$.

The most encouraging findings during this experiment, however, is that the positive effect of somatic hypermutation continues for $r \geq 13$. This may also be seen in figures 6.13 and 6.14 on page 51. Figure 6.13 and Figure 6.14 on page 51 also show the behavior of the AIS when *both* thymus behavior and affinity maturation was utilized. None of the figures show any decrease in the level of $\overline{\Delta A^+}$ where $r \in [6, 7]$ when comparing Figure 6.13 and Figure 6.14 with Figure 6.9 on the preceding page and Figure 6.10 on the facing page, respectively. On the other hand, by comparing the same figures in the region $r \geq 13$, a slight increase is noticed when thymus behavior *is* used.

Figure 6.15 on page 51 shows the effect of using affinity maturation combined with thymus behavior at the data set level when $r = 16$, $T = 50\,000$ and $T_{\text{death}} = 250\,000$. It clearly

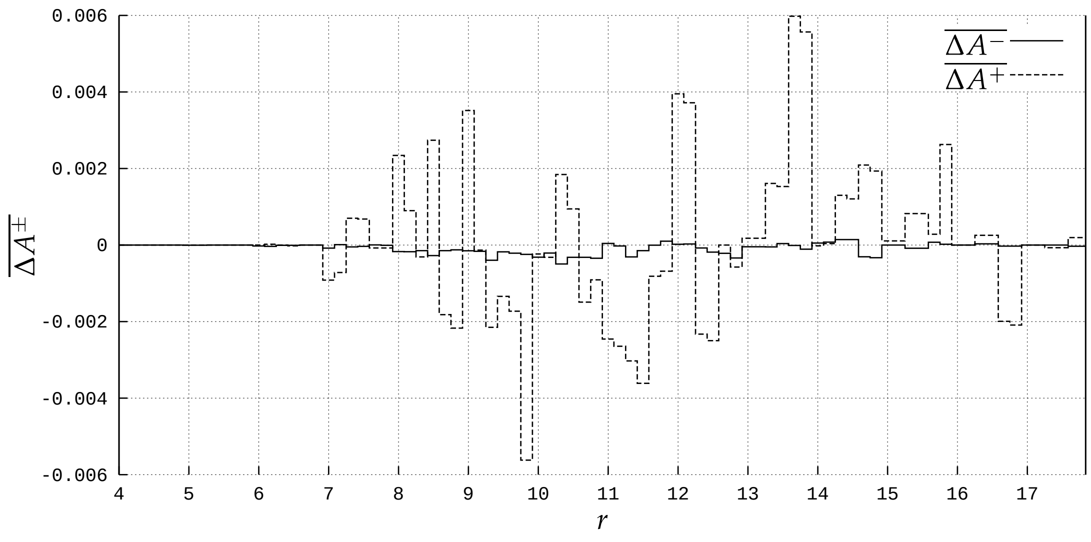


FIGURE 6.12: How $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 25\,000$, $(\alpha^{+}, \alpha^{-}) = (1.333, 0.8)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The higher the value of both $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$, the higher the performance gain from using somatic hypermutation. No thymus behavior was utilized, but affinity maturation was used.

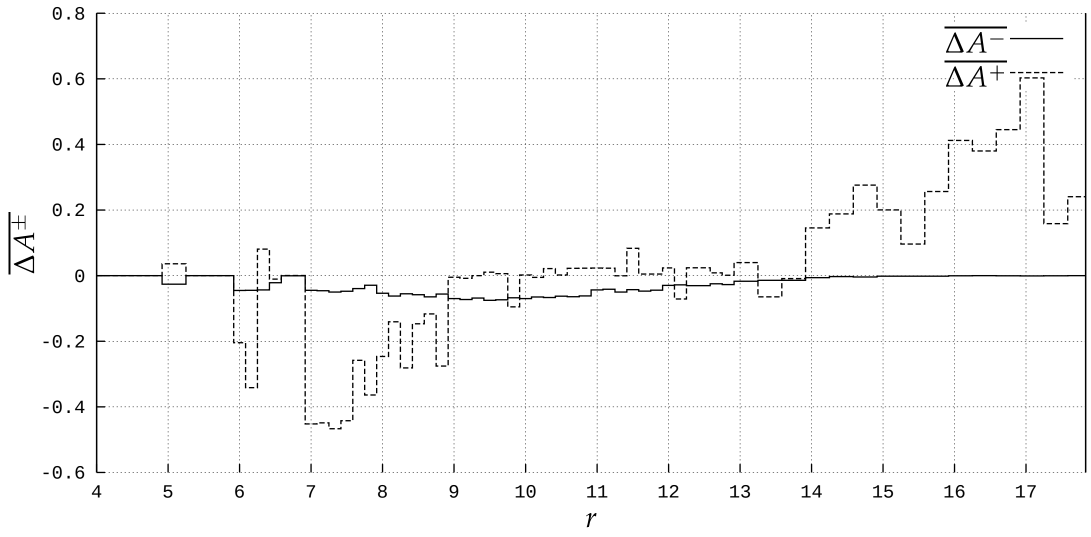


FIGURE 6.13: How $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^{+}, \alpha^{-}) = (1.2, 0.8)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The higher the value of both $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$, the higher the performance gain from using somatic hypermutation. Both thymus behavior and affinity maturation were utilized.

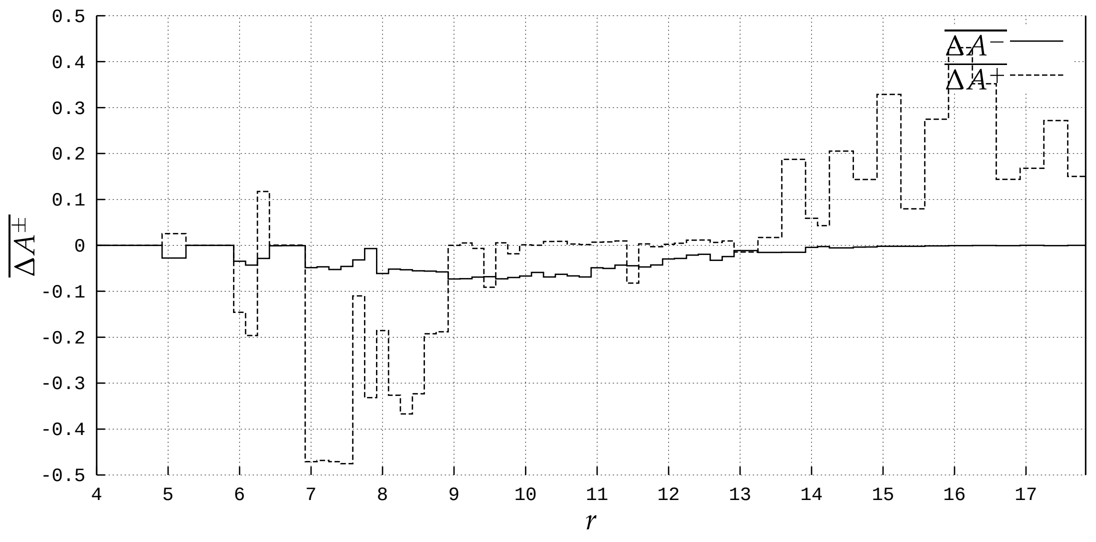


FIGURE 6.14: How $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$ vary when somatic hypermutation *is* and *is not* used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^{+}, \alpha^{-}) = (1.333, 0.8)$, $\xi = 10$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The higher the value of both $\overline{\Delta A^{+}}$ and $\overline{\Delta A^{-}}$, the higher the performance gain from using somatic hypermutation. Both thymus behavior and affinity maturation were utilized.

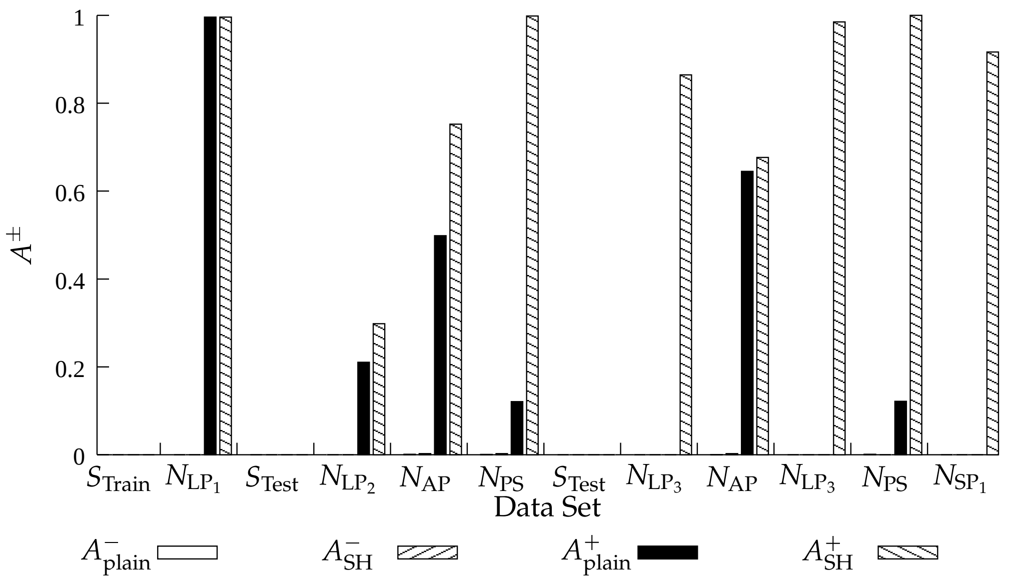


FIGURE 6.15: How A^{-}_{plain}, A^{-}_{SH}, A^{+}_{plain} and A^{+}_{SH} for each data set vary when somatic hypermutation combined with affinity maturation *and* thymus behavior *is* and *is not* used. The parameters used for this simulation were the same as the ones used in Figure 6.13 on the facing page, with the combination of $r = 16$, $T = 50\,000$ and $T_{\text{death}} = 250\,000$.

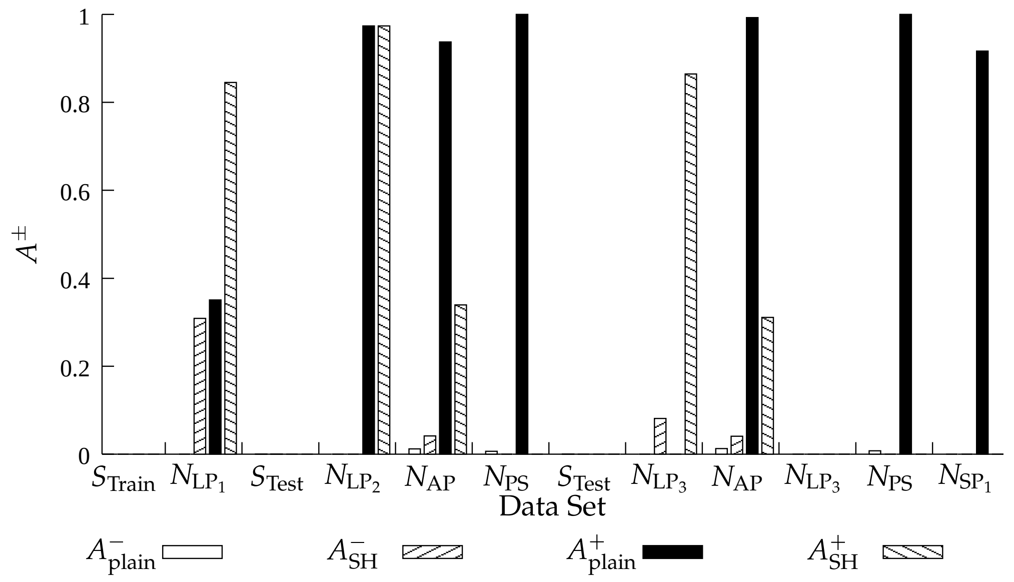


FIGURE 6.16: How A^-_{plain}, A^-_{SH}, A^+_{plain} and A^+_{SH} for each data set vary when somatic hypermutation combined with affinity maturation *and* thymus behavior *is* and *is not* used. The parameters used for this simulation were the same as the ones used in Figure 6.15, except that $r = 7$.

shows how the AIS is able to detect several nonself *agents* in all the nonself data sets when somatic hypermutation *is* used, but that it misses all nonself agents in some of the data sets and generally has a low true alarm rate when somatic hypermutation *is not* used. It should also be noted that with this high value of r, the AIS does not match any of the strings in S_{Test} neither *with* nor *without* somatic hypermutation, and it generally has a very low number of false alarms.

Figure 6.16 shows the effect of utilizing affinity maturation and thymus behavior compared to not using these mechanisms. One can see that when somatic hypermutation is utilized, the AIS performs worse over time than when it is not. Note that the AIS is exposed to the data sets in the order they are presented in the figure, from left to right.

To give an overall impression of the behavior of the AIS under different conditions, figures 6.17 to 6.19 are included. Figure 6.17 on the facing page shows how A^+ and A^- vary when no somatic hypermutation is used and $(\alpha^+, \alpha^-) = (1.2, 0.8)$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. There is a clear increase in the number of true alarms as the value of r gets higher in the range $r \in [4, 8)$. In the range $r \in [8, 12]$ the true alarm rate A^+ is at its maximum, while it decreases as r gets higher when $r \geq 13$.

Figure 6.18 on the facing page also shows how A^+ and A^- vary, but *with* somatic hypermutation and affinity maturation. The differences is notably that A^+ does not reach its maximum until $r = 9$ and that the significant decrease does not start until $r = 14$.

Figure 6.19 on page 54 shows how A^+ and A^- vary when somatic hypermutation, affinity maturation and thymus behavior is utilized. The trends in this figure are very similar to those in Figure 6.18, but with a higher values of A^+ when $r \geq 14$.

Figure 6.20 on page 54 shows how A^+ and A^- vary when *no* somatic hypermutation is utilized and with *costimulation delay* $\gamma = 25\,000$. Most notably is the pyramid shape of

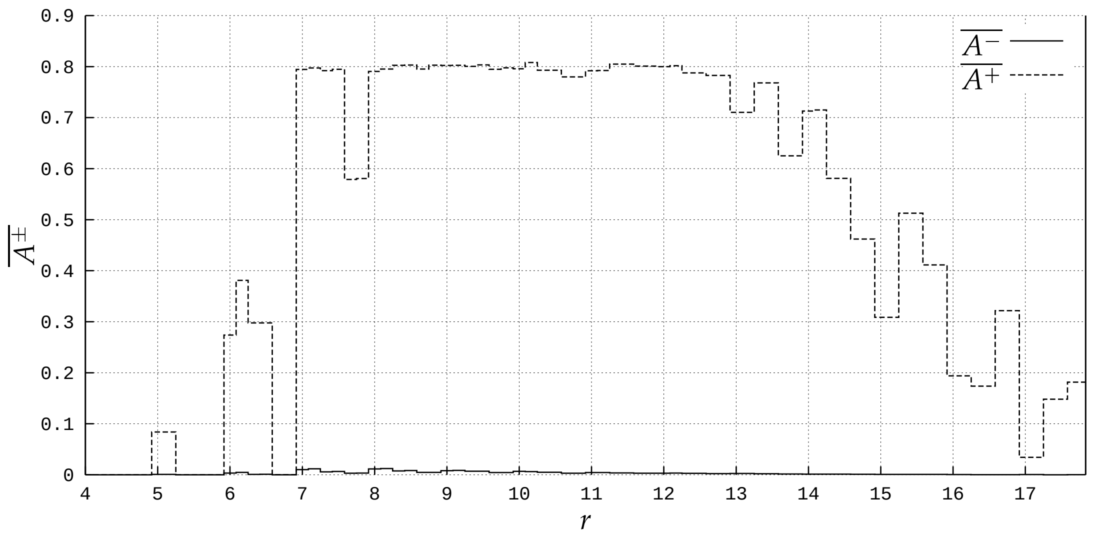


FIGURE 6.17: How $\overline{A^+}$ and $\overline{A^-}$ vary when no somatic hypermutation is used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^+, \alpha^-) = (1.2, 0.8)$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The same parameters were used in Figure 6.9 on page 48. No thymus behavior was utilized.

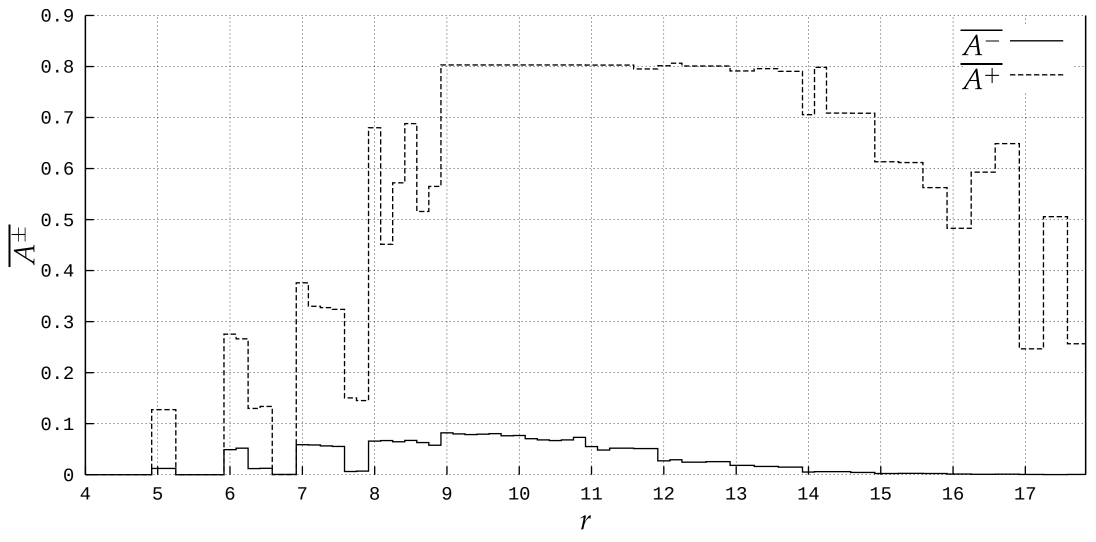


FIGURE 6.18: How $\overline{A^+}$ and $\overline{A^-}$ vary when both somatic hypermutation and affinity maturation are used. The parameters used for this simulation were $\gamma = 0$, $(\alpha^+, \alpha^-) = (1.2, 0.8)$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The same parameters were used in Figure 6.9 on page 48. No thymus behavior was utilized.

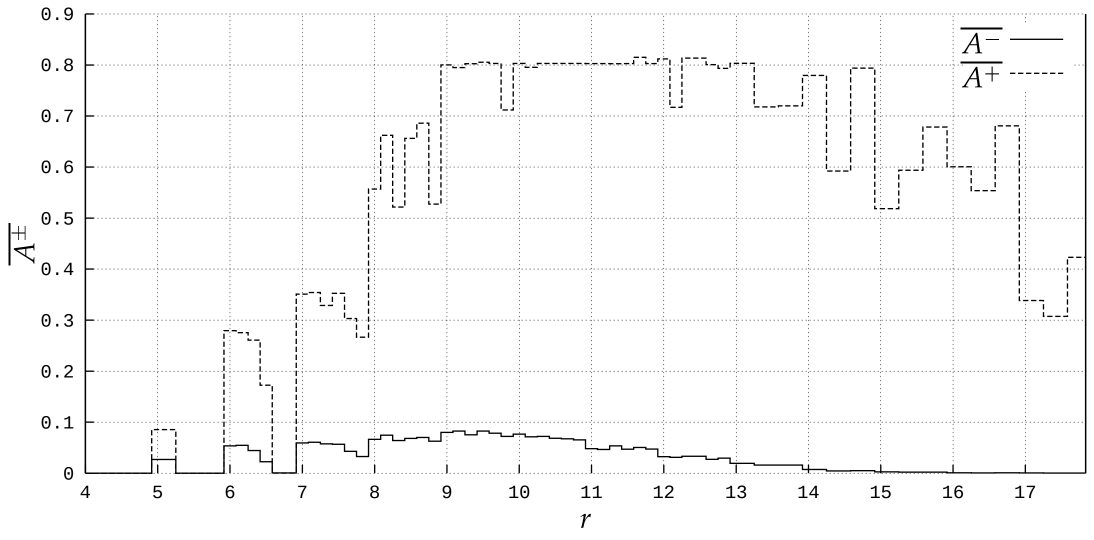


FIGURE 6.19: How $\overline{A^+}$ and $\overline{A^-}$ vary when both somatic hypermutation and affinity maturation are used combined with thymus behavior. The parameters used for this simulation were $\gamma = 0$, $(\alpha^+, \alpha^-) = (1.2, 0.8)$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The same parameters were used in Figure 6.9 on page 48. No thymus behavior was utilized.

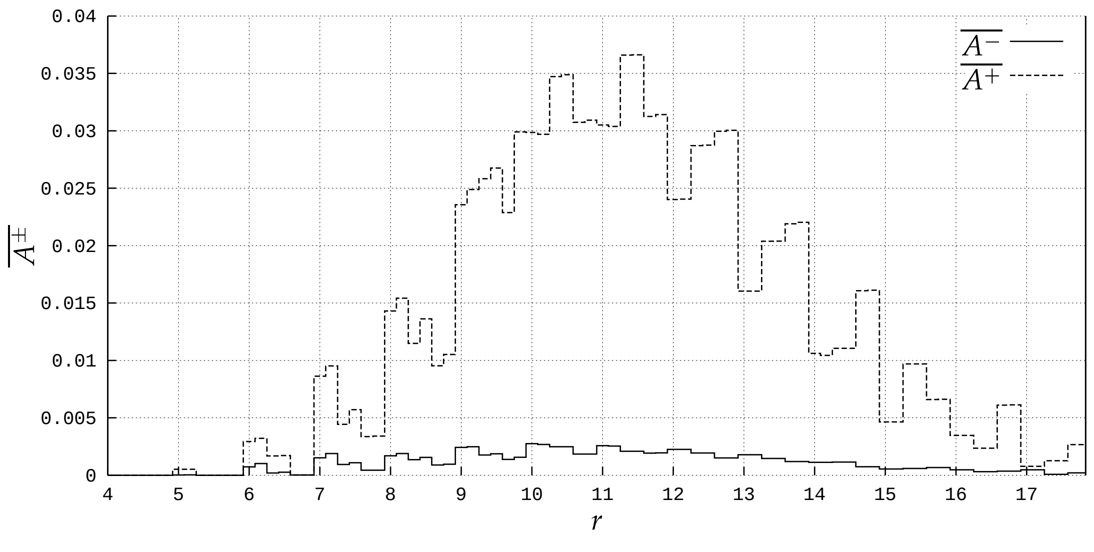


FIGURE 6.20: How $\overline{A^+}$ and $\overline{A^-}$ vary when no somatic hypermutation is used. The parameters used for this simulation were $\gamma = 25\,000$, $(\alpha^+, \alpha^-) = (1.2, 0.8)$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The same parameters were used as in Figure 6.9 on page 48. No thymus behavior was utilized.

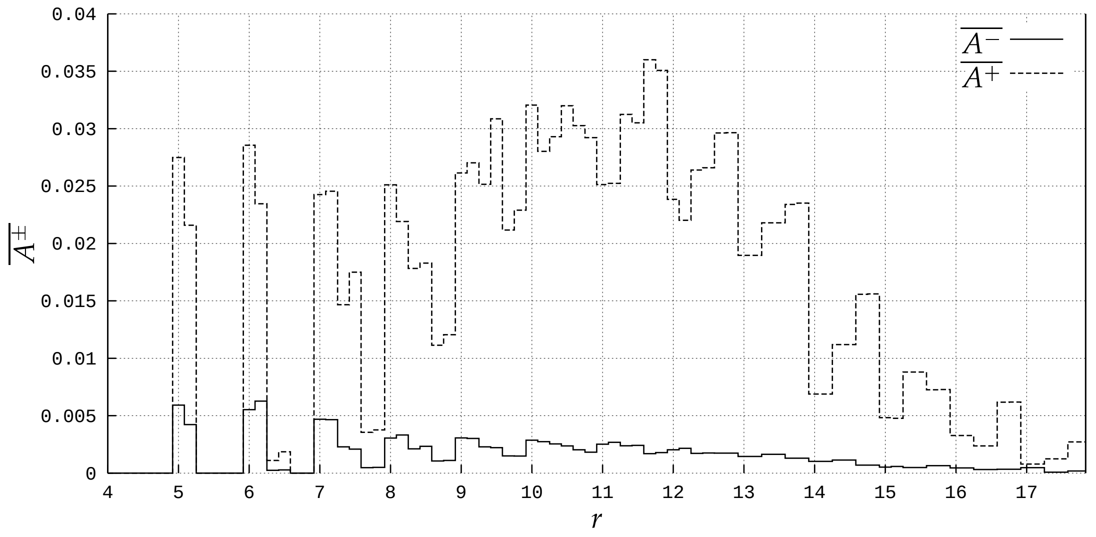


FIGURE 6.21: How $\overline{A^+}$ and $\overline{A^-}$ vary when no somatic hypermutation is used, but thymus behavior is utilized. The parameters used for this simulation were $\gamma = 25\,000$, $(\alpha^+, \alpha^-) = (1.2, 0.8)$ and the values for T and T_{death} are ordered as shown in Table 6.4 on page 41. The same parameters were used as in Figure 6.20.

the alarm rate level with a maximum when $r \in [10, 11]$ and the scale of the $A^{\pm}$ axis which is extremely small compared to the graphs where $\gamma = 0$. The number of false alarms is also at its maximum in the region where $r \in [10, 11]$. Most of these features are present in Figure 6.21 too, except that when $r \leq 9$ the thymus behavior seems to increase the number of true alarms quite dramatically, while there is a slight increase in the number of false alarms.

Figure 6.22 on the following page shows how A^-_{plain}, A^-_{SH}, A^+_{plain} and A^+_{SH} vary at the data set level, when the costimulation delay $\gamma = 25\,000$. A persistent feature present in most of the simulations where $\gamma = 25\,000$ during the second experiment is that the even though there are low values for $\overline{A^{\pm}}$, there are higher values for A^+ than for A^-, *and* there are some true alarms occurring while monitoring each *nonself* data set *except* during the second occurrence of N_{LP_3}.

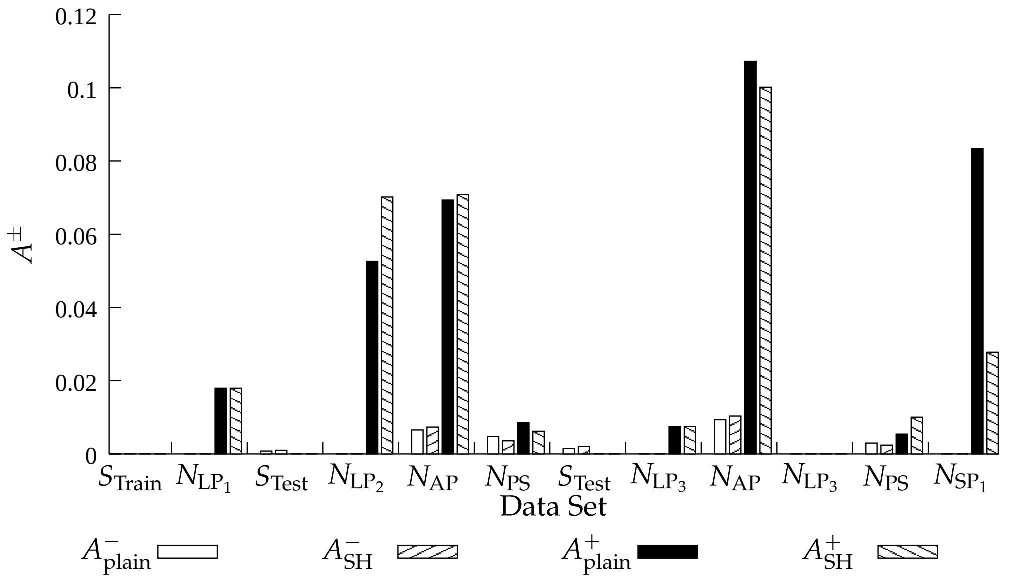


FIGURE 6.22: How A^{-}_{plain}, A^{-}_{SH}, A^{+}_{plain} and A^{+}_{SH} for each data set vary when somatic hypermutation, combined with affinity maturation *and* thymus behavior, *is* and *is not* used. The parameters used for this simulation were $\gamma = 25\,000$, $r = 10$, $T = 25\,000$ and $T_{\text{death}} = 250\,000$.

CHAPTER 7

DISCUSSION

The DAIS is a working prototype of a AIS-based NID. While it was used solely for controlled simulations in the experiments presented in Chapter 6, it is also able to function as a real-life AIS-based NID system by monitoring network traffic dumped by *TCPdump*.

7.1 LIMITATIONS OF THE EXPERIMENTS

The simulated NID was monitoring *TCP SYN* packets only. This is an important limitation, in that there is much more data being sent over the TCP in general than just SYN packets. In addition, it is very limiting for a NID system to only monitor a single protocol, as mentioned in Section 3.4.

The exclusion of some types of network traffic, such as, e.g., HTTP and FTP server traffic, in the datasets used for the simulations, was intended. This limitation was introduced due to the continuously changing array of new clients connection to the services. With the *peptide* encoding used in DAIS, this kind of "randomness" would probably increase the number of false alarms to a level where the introduction of such a system would have no use. On the other hand, it might be possible in the future to find ways in which AIS-based NID systems may monitor such services too.

Another limitation, or constraint, is that the results presented in Chapter 6 are representative for how the AIS behaves *the data sets used*. Thus, the discussion below will be based on the assumption that these data sets are representative for real-world network traffic. As will be discussed in Section 7.6, one of the apparent problems when performing NID research is that acquiring good data sets is a difficult task.

Finally, the parameters used for the experiments clearly defines some of the limitations of this experiment. First of all, only two different values were used for the *costimulation delay* γ, namely 0 and 25 000. In addition, these choices of γ are very far apart when counted in time units, but the idea behind using $\gamma = 25\,000$ is that it models the worst case of how the AIS will behave when an external source may provide *costimulation* within 1 day. Also, the maximum sizes of the subpopulations were kept constant, with the exception of the number of *immature detectors*, D_{Imax}. The number of *mature detectors* $D_{\mathrm{Nmax}} = 70$ and *memory* *detectors* $D_{\mathrm{Mmax}} = 100$ were fixed during the simulations, while D_{Imax} was changed from 40 in the first experiment to 30 in the second one. There may be some interesting characteristics of the AIS that would be shown if the parameters mentioned above were changed, but due to the huge sizes of the experiments, some limitations had to be introduced.

7.2 Interpretation of the Results

There were several trends present in the results presented in Chapter 6. One of the most important ones for answering the hypothesis stated in Section 4.5.1 is that the use of *clonal selection* combined with *somatic hypermutation* to model *affinity maturation* in the AIS certainly *affects* the systems ability to distinguish between *self* and *nonself*.

When interpreting the results of the experiments, it is important to mention what is the wanted behavior of the AIS as a NID. The ideal behavior would be if the AIS would recognize all connections, and also discriminate effectively between self and nonself connections, so that the number of true alarms is as *high* as possible, and that the number of false alarms is as *low* as possible.

7.2.1 Somatic Hypermutation

The trend most stable throughout the graphs is the added number of false alarms where $r \in [6, 13]$ when utilizing *somatic hypermutation*. This characteristic did not change much whether *affinity maturation* or *thymus* behavior was utilized. The number of false alarms was *not* significantly higher when using somatic hypermutation than not in the area where $r \leq 6$ and $r \geq 14$.

The reason there is no notable difference when $r \leq 6$ is probably that the AIS performs very bad for those values of r in general. As mentioned in Section 4.3.1, lower matching thresholds, r-values, during tolerization result in more general detectors. But, with too low matching thresholds, the AIS will need an extreme amount of retries to generate each valid *detector*; i.e., due to *negative selection*, most of the *immature detectors* die before they become *mature*. By evaluating $R_{1/2}(r, \ell)$ from (4.12) with $r = 6$ and $\ell = 49$, one can see that the probability of a match between a newly created immature detector (which is randomly created) and a randomly chosen bit string is 30.9%; and with $r = 4$, this probability has increased to 82.1%. This reflects the fact that because of the low r-values compared to the total length of the *peptides*, there is an increased probability that other peptides (*self* or *nonself*) contains a matching sequence of bits. This, of course, is also related to the use of alphabets with cardinality $m = 2$. To become a mature detector, however, an immature detector has to avoid matching *any agents* during tolerization, and as the probabilities above show the probabilities of *not* matching a detector *once* is very low. On the other hand, it should be noted that the self set is not as uniformly distributed as completely random strings. But, for somatic hypermutation to have any effect at all, there need to be *some* mature detectors present in the population D, so if $D = \emptyset$ *no* connections will be detected.

The few occurrences of gains in the false alarm rate when $r < 6$ is probably also related to the above mentioned fact. As there probably are few *valid detectors* in the population of detectors when such low values of r are used, any activation is likely to trigger a *clonal proliferation* of *memory detectors* which, in turn, are instantly activated when recognizing nonself peptide.

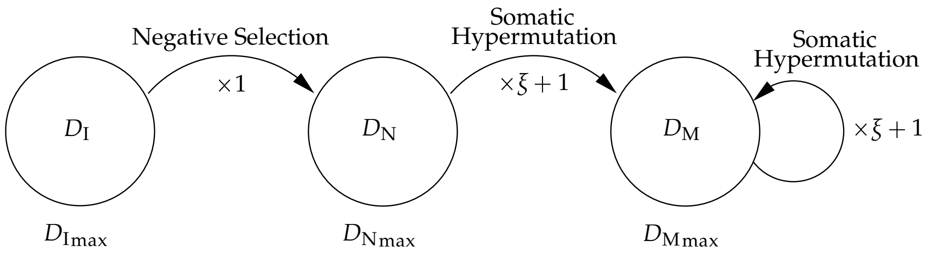


FIGURE 7.1: The development of the population of detectors D. The circles represent the subpopulations, where D_I consists of immature detectors, D_N consists of mature detectors and D_M consists of memory detectors. ξ denotes the number of clones which are created during somatic hypermutation.

The number of false alarms introduced by utilizing somatic hypermutation increases from $r = 6$ to $r = 9$ and decrease from $r = 10$ to $r = 14$. This is probably caused by the way affinity maturation is implemented in DAIS. As mentioned earlier, two different approaches to somatic hypermutation were implemented. In the first one, which utilize affinity maturation, the mutated daughter cells of a detector which exhibit higher affinity to the agent that activated become memory detectors. In the second implementation of somatic hypermutation, *all* daughter cells of the cloned detector become memory detectors. This means that no matter which of the two somatic hypermutation implementations are used, the cloned daughter cells become memory detectors immediately, without going through maturation, in the form of negative selection. In addition, because they are memory detectors, anything they match will cause an activation. When short contiguous matching regions are used, e.g. $r = 10$, the probability of creating a clone which binds (with high enough affinity) to other peptides than the originally detected agent is increased. The cause of the high false alarm rates may be that these clones will possibly also detect self.

If the above phenomenon is the only one affecting the false alarm rate when *somatic hypermutation* is utilized, the number of false alarms should be higher for $r = 6$ than for $r = 9$, which it is not (see, e.g., Figure 6.18 on page 53). A probable explanation for this is that the rate of false alarms introduced by somatic hypermutation in the region where $r < 9$ is reduced by the lower number of alarms in general (both true and false) in that region. Thus, this trend in alarm rates seems to be caused by a combination of the higher probability of introducing *autoimmune* detectors, and that too low matching thresholds make it difficult—it requires more retries—to create a detector set D which effectively differentiates between *self* and *nonself*.

The negative effect on the rate of true alarms when $r < 9$ is probably caused by another related phenomenon. Figure 7.1 shows how detectors over time are moved from one subpopulation to another in the AIS. First, when the system starts its training period, to generate a set of valid detectors, the AIS comprises D_Imax randomly generated immature detectors, so that $|D_\mathrm{I}| = D_\mathrm{Imax}$, $|D_\mathrm{N}| = 0$ and $|D_\mathrm{M}| = 0$. As discussed above, with lower values of r, the AIS requires more retries to generate each mature detector by utilizing *negative selection*. However, when a mature detector is activated and receives *costimulation*, it is transfered from D_N to D_M and undergoes somatic hypermutation; thus, D_N is reduced by 1 mature detector while $1 + \xi$ memory detectors are inserted into D_M. Even though the mature detectors require multiple matches within a limited time period before they are activated, with low r values they are likely to be transfered into D_M faster than new mature detectors are generated during negative selection. Hence, if a sufficient number of nonself agents are detected in a short time, $|D_\mathrm{N}|$ may become 0. This effect, combined with the fact that memory detectors are activated by a single match, and thus undergo somatic hypermutation—where the LRU memory detectors are replaced by newly generated ones—may lead to the evolution of a population consisting of detectors with very little diversity, and hence little ability to detect nonself agents. This factor seems to be the reason that the AIS performs worse when using somatic hypermutation than when not using it, for $r < 9$, and offers a plausible explanation to the negative trend that was seen throughout the results when $r < 9$. This can be seen at the data set level in Figure 6.16 on page 52.

The positive effect of *somatic hypermutation*, seen where $r \geq 14$, is likely due to the effect somatic hypermutation and *affinity maturation* have in the *IS*. When a *detector* is activated, by recognizing a *nonself agent*, it undergoes *clonal proliferation*, with a high probability of mutation in the daughter detectors' recognition regions. The daughter detectors compete with other detectors and clones to bind with nonself agents. The higher the affinity between a detector and monitored *agents*, the more likely it is to be activated, and thus cloned again. It also seems natural that there is a lower number of false alarms while using somatic hypermutation when $r > 14$, as this means that there is only a low probability that a single point mutation should enable a cloned detector to recognize a lot of different *self agents*.

The effect of the costimulation delay γ observed throughout the results first of all seems to reduce the fraction of both true and false alarms dramatically, as can be seen in Figure 6.20 on page 54. Still, the AIS was able to detect intrusions in almost all the data sets during the second experiment, see Figure 6.22 on page 56. The exception was the second time during each simulation that the AIS was exposed to agents in the N_{LP_3} dataset. This seems to be caused by the way binding between detectors and agents is implemented in DAIS. When a *memory detector* is activated, it binds to the activating agent. If the same detector matches any other agents while it waits for its *costimulation*, it is not activated again. Hence, no true alarm is registered either. This seems to cause the detectors to be bound to agents from the first encounter with N_{LP_3} the second time that data set is encountered.

Since detectors that do not receive costimulation die, the use of costimulation may be regarded as a form of *reinforcement learning*. When the costimulation delay, which plays the role of a reinforcing signal, gets longer, the learning of the AIS gets slower. This would explain the dramatically lower true and false alarm fractions when $\gamma = 25\,000$ than when $\gamma = 0$.

7.2.2 Thymus Behavior

The exploration of the *thymus* behavior, as defined in Section 5.4.3 has not been the primary focus of this work. However, it is an interesting mechanism that deserves a closer look. By the experimental results presented in Chapter 6, it seems that the thymus behavior, as modeled herein, does not affect the overall operation of the AIS much. However, under some circumstances, it causes a notable difference. The difference is very clear when comparing Figure 6.21 on page 55 where thymus behavior *is* used to Figure 6.20 on page 54 where it is *not* used. The trend is that when $r \leq 9$, the thymus behavior seems to add a significant number of true alarms, but also a slight increase in false alarms. This trend is probably caused by the fact that with low r-values it is difficult to generate a detector set with a sufficient *cover*, *but* with the utilization of the implemented thymus behavior it seems likely that several *self agents* are matched by either *mature* or *memory detectors* and thus will not cause *apoptosis* among the *immature detectors*.

7.2.3 Consequences

What follows is a summary of the above discussion and some of its consequences. First of all, it should be noted that some of the results presented and discussed herein seems to be highly dependent on the way certain IS-inspired mechanisms are implemented in DAIS. For example, as has been shown above, the AIS is very sensitive to variations in r, the *r-contiguous match* matching threshold. By replacing parts of the AIS with other algorithms, the results may be quite different.

The results indicate that when *detectors* with a high level of specificity, $r \geq 14$, are used, the utilization of *clonal proliferation* combined with *somatic hypermutation* enhances the ability of the AIS to detect a higher fraction of *nonself* agents without adding a significant number of *self* activations. This finding should answer the hypothesis presented in Section 4.5.1, but not without exceptions. These exceptions will be summarized below.

With low r-values, $r < 9$, the use of somatic hypermutation dramatically reduces the number of true alarms. This seems to happen because there is not generated enough valid detectors through *negative selection* while at the same time somatic hypermutation leads to the emergence of a population with extremely low diversity. The process of *affinity maturation* may be seen as a *Darwinian process* of *variation* and *selection*, but the way it is implemented in DAIS, it does not alone introduce enough variation into the population of detectors for a population with an adequate *cover* to emerge. However, if affinity maturation is utilized in populations where a sufficient number of new detectors are constantly generated through *negative selection*, it will lead to the emergence of a population with a larger cover than when it is not utilized.

Based on the above findings, it seems likely that *with* somatic hypermutation one will be able to either achieve a higher level of coverage with the same *population size* as when it is *not* used, or to keep the same *level of coverage* by using a smaller population. As mentioned in Section 4.5.1, the gain of using higher r-values is that then the AIS needs fewer retries to generate a valid detector, while the cost is that there is needed a larger population to achieve the same level of coverage. With somatic hypermutation it should hence be possible to generate a population of valid detectors with higher specificity faster than without. These aspects are all concerning the resource usage of the AIS. In short, the findings herein suggest that somatic hypermutation may be used to decrease the resource consumption by a AIS. The importance of this will be discussed further in Section 7.4.2.

Another finding, not related to somatic hypermutation, should also be noted. In (Hofmeyr and Forrest 2000) the ARTIS architecture is presented along with some results of simulations performed with an AIS implementation called LISYS. Although the implementation by no means is identical with DAIS, it implements some of the same IS-inspired mechanisms—but, e.g., not somatic hypermutation or affinity maturation. The simulations were performed using the same data sets as used herein, although probably not in the same order, and with a *costimulation* delay of 25 000 and a population size of 100 detectors per location in their simulated network consisting of 50. The simulations suggested an optimal operation with $r = 12$, while under almost the same conditions, the results presented herein, see Figure 6.20 on page 54 suggest an optimal value of $r = 10$ or $r = 11$. Later, when performing new tests with LISYS with a newer data set, Balthrop, Forrest, and Glickman (2002) reported a lower optimal value for $r = 10$.

7.2.4 Possible Improvements

The findings in the interpretation of the results suggest some improvements of the *somatic hypermutation* algorithm implemented in DAIS. It seems that if the clones generated during somatic hypermutation had to undergo some kind of *negative selection*, then somatic hypermutation would hopefully not introduce as many false alarms as now, when $r < 14$. However, for values of $r < 9$ there would probably still be a problem with somatic hypermutation destroying the diversity of the population, which in turn causes a strong decrease in the number of true alarms reported.

Another improvement would be to find a better way to handle the binding of an *detector* to an *agent* when *costimulation* delays $\gamma > 0$ are used. One of the apparent problems with the current implementation is that *one* detector binds to *one* agent, and since the AIS should use as few resources as possible, the suggestion is put forward to let detectors that are bound to agents be "activated" multiple times. This way, each detector will represent an unlimited array of such identical detectors. The problem this introduces is that of how to determine which one of the activating agents should be used to test for *affinity* during *affinity maturation*. One simple solution would be simply to use the first activating agent, but there may be better ways.

7.3 Possible Sources of Error

The *TCPdump* program, which was used for dumping the network traffic while creating the data sets, can become overloaded and drop packets, although the possibility of this happening is reduced by using low data rates.

7.4 The Role of Artificial Immune Systems in Network Intrusion Detection

The exploration of AIS-based NID systems has just been started, so there is a lot of work to be done before one knows enough to safely deploy AIS-based NID systems in *real-world* LANs. The results presented herein indicate that it may very well be *tested* in real-world *networks*, but that it should not be used as a primary line of defense against network intrusions.

As mentioned in Section 3.6 one of the current trends in LAN technology is that organizations change from *broadcast* to *switched* LAN technology. One of the great features of DAIS is that even though it is distributed, it does not depend on any information being sent between the nodes in the LAN to describe the problem other than the broadcasted network traffic. The nodes are still able to cooperate to solve the overall problem—to detect anomalies in the network traffic—as they all get the same information about the status of the LAN. In a switched network, this will not be possible by with the *peptides* currently used by DAIS. It should be noted that there are mechanisms implemented into DAIS which easily enables the simulation of switched LANs, but the peptide-problem remains. Dasgupta and Forrest (1996) has shown that their classifying AIS showed good abilities of novelty detection in time series data. This should be encouraging with respect to designing an effective peptide-representation for use in switched networks, as one of the possible ways to distribute information of the overall status of a switched LAN seems to depend on the distribution of different frequencies of various events. Another, banal, solution would be to alter the behavior of the network switch to broadcast all network information, just like a network hub. This may actually be done with a technique used in security cracking.

The field of ID has evolved rapidly in the last few years. Still, several important issues remain to be solved. Kemmerer and Vigna (2002) present some of the open issues in the field of ID:

> First, detection systems must be more effective, detecting a wider range of attacks with fewer false positives. Second, intrusion detection must keep pace with modern networks' increased size, speed, and dynamics. Finally, we need analysis techniques that support the identification of attacks against whole networks.

The role of AISs applied to NID with respect to these issues will be discussed below.

7.4.1 Performance

Simply *detecting* a variety of *events of interest* is not enough. An efficient IDS should also keep up with the stream of data generated by high-speed networks and high-performance network nodes. As stated by Kemmerer and Vigna (2002), the combination of increasingly fast networks and faster network nodes has the effect that system administrators who try to analyze their log files are confronted with mountains of data.

The results presented herein suggests that DAIS is well suited for this kind of task. As it is *distributed*, with small subpopulations that work together to detect network intrusions, the workload of monitoring the network traffic is divided on the number of hosts performing the task. The fact that no communication is required between the different nodes makes DAIS *robust* too, since even if some locations, or nodes, stop running the performance of the AIS is only gradually reduced.

In addition, the findings of the experiments performed in this thesis work suggest that with *somatic hypermutation* it is possible to achieve that same level of *coverage*—and hence the ability to detect *nonself*—with less *detectors*

7.4.2 Striving for Autonomy

Even though DAIS seems well suited to the task of coping with the vast stream of data generated by high-speed networks, there are some issues related to performance and efficiency worth discussing.

The implemented AIS-based NID, DAIS, has been used to simulate two different scenarios with respect to the length of the *costimulation delay* γ. When $\gamma = 25\,000$ it is meant to model a LAN where a human operator provides the second signal which an *activated detector* will depend on. If the *detector* does not receive *costimulation*, it dies. Thus, the default assumption made by the AIS is that the activation is a false positive. If, on the other hand, it receives costimulation, it will live and undergo *somatic hypermutation* if it is utilized.

This behavior of assuming the activation was a false positive is not accidental. This policy is used to strengthen the security of the AIS. Since DAIS is (or simulates that it is) *distributed* over several hosts, the costimulation signal provided by a human operator must be sent over some interface to the host where the particular detector is awaiting costimulation. If the policy had been to assume that all activations are true positives—and hence, let all activated detectors live, unless costimulation is received—an attacker of such a NID system might send false costimulation packets to the system to create an artificial tolerance of nonself network traffic.

On the other hand, in an ideal situation the NID system will not depend on a human operator, but operates autonomously. As many of the network intrusions seen today are performed from various compromised hosts, the development of a NID system which, once installed, does not depend on human expertise, will probably reduce the number of this kind of attacks.

One of the really difficult kinds of attacks, which an AIS-based NID system with such qualities probably would be able to stop—or to dramatically reduce the effect of—is *distributed denial of service (DDOS)* attacks. DDOS attacks usually utilize an array of compromised hosts to execute copies of programs that, when used together, are able to render the victim hosts, or complete networks, unusable due to resource starvation or bandwidth consumption; i.e., they run out of RAM, CPU-cycles, *bandwidth* or other resources needed to run their services (Northcutt et al. 2001, pp. 189–232). As these attacks prove very difficult to handle when they are first executed, it seems that if one could avoid that the array of computers used under such attacks got compromised in the first place, it perhaps reduce the spread of such attacks. One solution could thus be to design an AIS which requires few resources based on the idea presented below.

A suggestion as to how an AIS may be designed to implement these features, is to combine a distributed AIS-based NID system, like DAIS, with another IS related idea, presented by Forrest et al. (1996). They present a way to perform *host-based anomaly detection* by monitoring *peptides* that represent short sequences of *system-calls* executed by processes. Their overall idea is to build up a separate database for each process, or program, of interest. Once a database is stable—i.e., no new sequences are added during normal operation of the corresponding process—the database may be used to monitor that process' ongoing behavior. The stored sequences of system-calls thus define the normal behavior of the process and, likewise, abnormal sequences indicate anomalies it the running process. The initial experiments performed by Forrest et al. (1996) suggest that with this *peptide encoding*, the implicit definition of self is stable during normal behavior of standard *Unix* programs.

By combining the above host-based anomaly detection method with DAIS there should be several important gains. First of all the host-based anomaly detection may be used to deliver *costimulation* for to activated *detectors*. This, in turn, would probably reduce the costimulation delay considerably. It is not realistic that the costimulation delay would be reduced to zero, but at least to a much lower level that 1 day. Perhaps it may be reduces to something like a minute, which approximately corresponds to 17 time units in the simulations performed herein. With this combination of host-based and *network-based anomaly detection* the learning in the AIS should be faster, and hence better performance with respect to the number of true and false positives is expected. This solution seems viable also by the fact that DAIS, as it is implemented today, requires very few resources.

When discussing autonomous behavior of NID systems, it seems natural to look at the possibilities the introduction of *automated response* in AIS-based NID systems might bring.

7.4.3 Automated Response

There are several aspects of *automated response* which make it worth striving for. First of all, it is cheap with respect to what resources an organization will have to spend to utilize it. Secondly, automated response is relatively easy to perform, compared responding to a possible intrusion manually. On the other hand the possibility of reacting to a *false positive* makes *active automated responses* hazardous. They might result in affecting innocent parties. Thus, it would be safer if one could utilize *passive automated responses*.

Somayaji and Forrest (2000) present a way of performing *host-based automated response*, by delaying *system-calls* in the OS. The computer program they describe, called *process Homeostasis (pH)*, is inspired by the homeostatic mechanisms which biological organisms use to stabilize their internal state. It is implemented as an extension to the *Linux* kernel. The program detects unusual behavior by monitoring changes in short traces of system-calls and slows down the system-calls of processes classified as anomalous.

A related idea, based on the same principle, is to modify the NIC-drivers of the *OS* to slowly stall detected network intrusions. There are other possibilities, such as blocking traffic from hosts that have been detected as an possible intrusion at the *firewall* level of a LAN. This response, on the other hand, is very aggressive and will possibly affect innocent parties.

7.4.4 The Worm Threat

A computer *worm* is a program that is able to spread itself from host to host in a network by automating intrusions. It is generally suggested that if you suspect your LAN to be under attack from a worm, you should cut off all connections to outside networks until the worm is isolated and removed. Thus, the spread of this kind of malicious code can severely damage a victim with regard to accessibility and time, which in turn may lead to great economic losses.

Staniford, Paxson, and Weaver (2002) argue that future, better engineered, Internet worms should be able spread to hundreds of thousands of hosts within minutes or even tens of seconds. One of the main problems with these worms is that they may be controlled, modified and maintained indefinitely; making it extremely difficult to detect these infective agents with *rule-based NID* systems. The use of *anomaly-based NID* systems—including DAIS—to prevent attacks of this kind seems very promising because they are able to detect malicious code never encountered before.

7.4.5 Non-detected Intrusions

There are several ways to trick a AIS-based NID system like DAIS to not detect a network intrusion. One of the obvious ways would be to perform the attack over a protocol not monitored, i.e., not over TCP in this case. Another way is to design an attack so that it is indistinguishable from *self*. This may be done simply by monitoring the normal—or self—traffic over time, and try to find weaknesses in the programs that service self-connections.

Yet another—but much more sophisticated—way to succeed in an attack on this kind of AIS-based NID system is to design an attack that is distributing its network traffic over such a long period of time that none of the *detectors* gets *activated*. This kind of prolonged attack could therefore be used against any kind of service in the network, even if they are monitored by the AIS-based NID system.

Some of the symptoms mentioned above is caused by the chosen *peptide encoding*. The lack of any frequency information seems to be one of the most important shortcomings of the current implementation. On the other hand, it has been shown (Kim and Bentley 2001) that finding good peptide encodings which takes traffic frequencies into account and work effectively with the *IS* inspired mechanisms used (such as *r-contiguous match* and *negative selection*) is difficult.

For example, some network attacks, called *SYN floods*, try to exploit a flaw in TCP itself. Figure 5.1 on page 28 shows how a *three-way handshake* is to be performed. During a SYN flood attack, the client sends lots of the *initial SYN* packets, but not the *ACK* packets expected in return after the server's *SYN/ACK* packet has been sent (Northcutt et al. 2001). Because the TCP is designed so that the communicating parties should wait when a possible error has occurred, the server will wait for the response for some time. If the initiating SYN packets are received frequently enough, this may lead to a resource starvation attack on the server, because it needs to reserve memory for all the possibly initiated connections.

Now, if the attacking host is *not* part of the set of hosts normally connecting to the host being attacked, this approach would probably be detected with the current *peptide encoding*, because of the high frequency of connections which would probably activate *some detector*, even if it is not a *memory detector*. On the other hand, if the attacking host *is* part of the set of hosts normally connecting to the host under attack, there is a low probability that *any* of the detectors will match the Ag *peptide* at all; therefore the attack would probably succeed undetected. This last kind of attack should not be difficult to detect, if one used some temporal and/or sequencing information in the peptide encoding.

7.5 Peptide Encodings Used by Others

Kim and Bentley (2001) argue that the *peptides*, used by Hofmeyr (1999), Hofmeyr and Forrest (2000) and herein, which is based on an encoding of *three-way handshakes* as strings of letters from an alphabet with cardinality $m = 2$ (i.e., binary) with length $\ell = 49$ is very limited in order to detect various types of network intrusions.

Kim and Bentley use a wider *definition of self* which takes into account 33 different parameters, derived from *TCPdump* data logs[1]. Their self definition includes both the source and destination *IP addresses* and *IP ports*, indications of known port vulnerabilities, indications of errors during the three-way handshake and *traffic intensity*. This resulted in encoding peptides as strings of length $\ell = 33$ consisting of symbols from an alphabet with cardinality $m = 10$, i.e., every element, e, of the string may be any value in the range $[0, 9]$.

Now, as was presented in Section 4.3.1 and Figure 4.4 on page 23, the *r-contiguous match* rule is quite insensitive, with regard to the probability of a match, to changes in ℓ, while it is very sensitive to changes in the values of r. Kim and Bentley (2001, p. 1334) made the assumption—by using (4.2)—that the probability for a match at any one single position in the string s was $p = \frac{1}{m}$; i.e., there is an equal chance that a symbol s_i in the string s is any one of the symbols $s_i \in [0, m-1]$. Then, Figure 7.2 on the next page shows how the values of r and ℓ affect the probability of a match between two randomly chosen strings of length ℓ with cardinality $m = 10$. Still, the r-contiguous match rule is little sensitive to changes in ℓ. On the other hand, it should be noted how the change of cardinality, m, affects the matching probabilities.

A consequence of using strings of cardinality $m = 10$ is that the number of possible

[1]Kim and Bentley (2001) used the "Information Exploration Shootout" NID data set, available from `http://iris.cs.uml.edu:8080/`. These data sets comprise 1 "baseline" (or self set) and 4 attack sets. The self set includes only normal traffic, while the attack sets include normal traffic *and* attack data. The attacks present are "IP spoofing attack", "guessing rlogin or ftp passwords", "scanning attacks" and "Network Hopping".

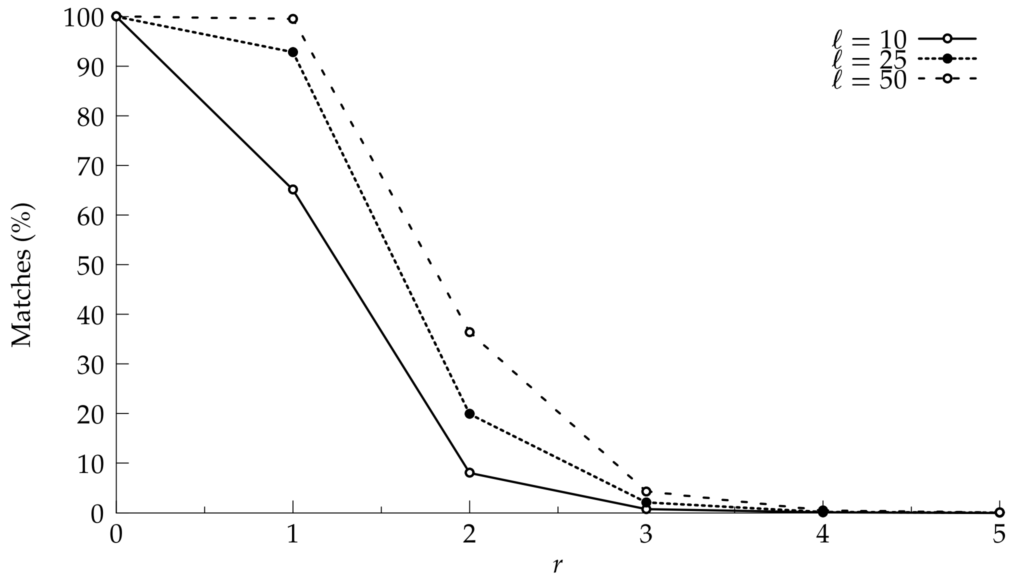


FIGURE 7.2: How the the choice of r and ℓ affects the probability of a match, using the contiguous match rule.

strings to construct is increased from 2^ℓ to 10^ℓ. This is a *huge* increase. The number of unique strings with $\ell = 33$ and $m = 10$ is $10^{33}/2^{49} = 5^{33}/2^{16} = 1.776\,36 \times 10^{18}$ times greater than the number of unique strings of $\ell = 49$ and $m = 2$.

Kim and Bentley report that they used two different r-values, $r = 4$ and $r = 9$. When $r = 4$ they were not able to generate one valid detector in 24 hours, but with $r = 9$ they were able to produce a valid detector in a reasonable time. They further report that with $r = 9$, the number of *detectors* needed to gain a good *nonself* detection rate is too high. Thus they argue that the *negative selection* algorithm is infeasible for the purpose of generating a set of valid detectors in a reasonable amount of time, and has a severe scaling problem.

Even though the above presentation of the extremely expanded search space during such an approach by no means is a proof of any kind, it suggests that Kim and Bentley (2001) may have chosen an inappropriate combination of peptide representation and matching rule to solve their problem.

7.6 Need for More Data Sets

Another important point to be made from the experiments performed is that there is a great need for more experimental data sets. To be able to perform reliable future experiments, there is a need for data sets that contain both different TCP traffic and perhaps also data from other protocols.

This requires a lot of work, as every connection should be correctly classified *a priori* of the experiments. This work will probably need to be performed manually. In addition, such data traces reveal how the networks where they are generated are organized. They may also reveal possible weaknesses in the hosts comprising those networks. Because of this, there are several *privacy concerns* to be made. If universities or other institutions are to provide such data sets to the public—mostly researchers and perhaps some commercial

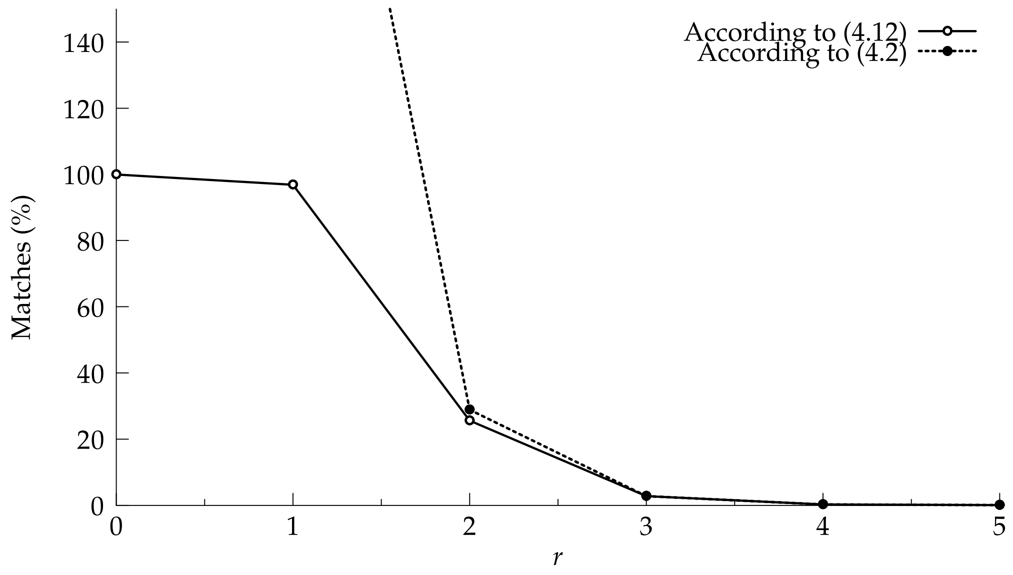


FIGURE 7.3: The difference between (4.2) and (4.12) in predicting the probability for a matching between two randomly chosen strings consisting of letters from an alphabet with cardinality $m = 10$, using the contiguous match rule when r changes. In both cases $\ell = 33$.

developers—there has to be some kind of *trust* between the parties. One solution might be to require that parties acquiring such data sets will have to sign some kind of agreement that the data will not be abused.

7.7 Evolution and Learning

By the concepts presented in this work, it should be apparent that the field of AISs is related both to A-life and *evolutionary computation (EC)*. Thus it seems natural to comment on some of the approaches used in DAIS which is also used in these research areas.

In the field of *evolutionary algorithms (EAs)*, a common approach to solving problems of different kinds is to maintain populations of individuals that represent candidate solutions. These populations usually undergo some type of evolution, in terms of mutations and genetic crossover, to hopefully arrive at better solutions (Michalewicz and Fogel 2000). There are also approaches to EAs which focus not on the individual level of a population but on the population as a whole.

In DAIS both approaches are used. For example, each location l in DAIS represents a subpopulation of *detectors D*. During *clonal proliferation* and *somatic hypermutation* some of the individuals of the population are cloned and their offspring are mutated. After mutation the each offspring is evaluated as possible solutions for the subproblem of best matching an *agent*. So during somatic hypermutation the focus is on the individuals of the population, while the performance of the DAIS in general is evaluated by the effect of the *whole population* working together. It should thus be noted that, since DAIS is *distributed* over several locations, with one independent—in the sense that no communication is used between the locations—subpopulation at each location, it may be argued that the performance of DAIS is evaluated on a super-population level.

It should also be noted that the exploration of the relationship between *learning* and *evolution* has received much attention in the field of EC. In DAIS this relationship is present in that *clonal selection* is the learning process used by the IS (Hightower, Forrest, and Perelson 1996).

7.8 Further Work

In Section 4.3.1 an expression (4.12) for the probability of a match between to randomly chosen strings of length ℓ, consisting of symbols from an alphabet with cardinality m, using the *contiguous match* rule with a matching constraint r was developed and analyzed. This expression was developed analytically and is a more correct replacement for the approximate expression (4.2) used in earlier work on AISs. Although some attempts has been done to formalize the theory behind AISs (Hofmeyr 1999), there seems to be a lot of work remaining to be done.

With regard to *somatic hypermutation*, now that the mechanism has been observed effectively applied to an AIS-based NID system, several smaller experiments should be performed to study its finer workings. Along with these future investigations, the improvements suggested in Section 7.2.4 on page 62 should be implemented to see if they will improve the effect of somatic hypermutation. Further work should also explore the effect of applying somatic hypermutation when the *Hamming match* rule is used.

Further work should also include testing DAIS in a real-world environment in real-time. As the implementation of an AIS has proved to be a nontrivial task, the release of DAIS under the GNU GPL suggests that others could either improve DAIS or investigate its source code to find inspiration and solutions for their own systems.

APPENDIX A

SOURCE CODE

Below, the *source code* for DAIS, the implementation described in this report, is presented. It is written in the *C* programming language.

The program compiled without errors and warnings using the *C* compiler from the *GNU Compiler Collection (GCC)* version 2.95.3 with the warning flags `-Wall` and `-Winline`. When compiling with GCC version 3.1, some warnings were issued because the concatenation of string literals with `__FUNCTION__` is deprecated in that version, but it still compiled nicely.

For optimizing the compiled program, with regard to speed and efficiency, the command line options `-O3`, `-D_GNU_SOURCE`, `-mcpu=athlon-xp`, `-march=athlon-xp`, `-malign\ -double` and `-funroll-all-loops` were used when calling the *C* compiler.

The implementation makes some use of the doubly linked lists provided by *GLib*. Thus, the program needs to be linked with this library. *GLib* is a general-purpose utility library, which provides many useful data types, macros, type conversions, string utilities, file utilities, a main loop abstraction, and so on. It works on many *Unix*-like platforms, *Windows*, *OS/2* and *BeOS*. *GLib* is released under the GNU *Library General Public License (LGPL)*.

In addition to the source for the AIS, the source for the program `fast_log_parse` is included at the end of this chapter. The program provides a fast way to parse and summarize a whole DAIS log file. This program was essential during the experiments performed as part of this work.

A.1 AIS.C

```
/* $Id: ais.c,v 1.28 2002/05/27 18:44:44 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif /* HAVE_CONFIG_H */
```

```
#include <stdlib.h>
#include <string.h>

#ifdef HAVE_FCNTL_H
#include <fcntl.h>
#endif /* HAVE_FCNTL_H */

#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif /* HAVE_UNISTD_H */

#include <sys/time.h>

#ifndef FALSE
#define FALSE 0                 /* This is the naked Truth. */
#endif /* ! FALSE. */

#ifndef TRUE
#define TRUE 1                  /* ... and this is the Light. */
#endif /* ! TRUE. */

#include "ais.h"
#include "network.h"

#define AIS_NODE_MAP_BUF_SIZE 1024

/* The node map should not be larger than 256, as the traffic was
   logged on a class C LAN. */
#define AIS_NODE_MAP_SIZE 256


/* Declare network_next_connection_func_t as typedef pointer to
   function that expects (stream as pointer to FILE, connection as
   pointer to network_connection_t) returning unsigned int. */
typedef unsigned int (* network_next_connection_func_t)
     (FILE *stream, network_connection_t *connection);

/* Declare agent_from_connection_func_t as typedef pointer to function
   that expects (agent as pointer to pointer to bitstring_t, len as
   const unsigned int, connection as pointer to const
   network_connection_t) returning void; */
typedef void (* agent_from_connection_func_t)
     (bitstring_t **agent, const unsigned int len,
      const network_connection_t *connection);

/* Create a pool for clonal proliferation. */
void
ais_clone_pool_create (ais_t *ais)
{
  unsigned int i;

  /* Allocate memory for CLONE_POOL. */
  ais->clone_pool = malloc (sizeof (*ais->clone_pool) * ais->config->clones);

  /* Check that the allocation was successful. */
  if (ais->clone_pool == NULL)
```

```
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  for (i = 0; i < ais->config->clones; i++)
    {
      /* The NULL (data) pointer is a request for generating a
         bitstring with 0s only. */
      (ais->clone_pool[i]).detector =
        detector_new (ais->config->len, NULL, DS_NONE,
                      ais->config->detector_activation_level_min);

      ais->clone_pool[i].affinity = 0;
    }

  return;
}

/* Create a pool for clonal proliferation. */
void
ais_clone_pool_destroy (ais_t *ais)
{
  unsigned int i;

  for (i = 0; i < ais->config->clones; i++)
    {
      /* Destroy the detector_t instances and free the resources
         reserved for them. */
      detector_destroy (ais->clone_pool[i].detector);
    }

  return;
}

/* Initialize an AIS instance. */
void
ais_initialize (ais_t *ais)
{
  unsigned int n;

  /* Allocate memory for an array of ais_node_t objects and check that it was
     allocated successfully. */
  ais->node_list = (ais_node_t *) malloc (sizeof (*(ais->node_list))
                                          * ais->config->nodes);
  if (ais->node_list == NULL)
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  /* Create a new allocator, for efficient memory handling when working with
     the GSList structure.   It also seems necessary to avoid memory leaks.
     Set the number of elements in each block of memory allocated to 1024. */
  ais->allocator = g_allocator_new ("AIS_detector_allocator", 1024);

  /* Initialize every newly allocated node. */
  for (n = 0; n < ais->config->nodes; n++)
    {
      node_initialize (&(ais->node_list[n]), ais->config);
    }

  /* If we are to simulate somatic hypermutation, we need an additional pool
     for clonal proliferation. */
  if (ais->config->somatic_hypermutation == TRUE)
    {
      /* Create and initizlize a pool with space for clonal proliferation. */
      ais_clone_pool_create (ais);

      /* Initialize every newly allocated node. */
      for (n = 0; n < ais->config->nodes; n++)
        {
          /* Create a link to the clone_pool, accessible from every node. */
          ais->node_list[n].clone_pool = ais->clone_pool;
        }
    }

  return;
}

void
ais_generate_node_map (ais_t *ais)
{
  FILE *stream = NULL;
  unsigned int done = FALSE;
  unsigned int i = 0;

  /* Allocate enough memory for the node map. */
  ais->node_map = (char *) malloc (sizeof (char) * AIS_NODE_MAP_SIZE);
  if (ais->node_map == NULL)
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  stream = fopen (ais->config->node_ids, "r");
  if (stream == NULL)
    {
      perror (PACKAGE ": "__FUNCTION__);
      exit (1);
    }

  /* Read all the lines of the file. */
  while (! done)
    {
      char buf[AIS_NODE_MAP_BUF_SIZE];
      unsigned int node;

      if (NULL == fgets (buf, AIS_NODE_MAP_BUF_SIZE, stream))
        {
          done = TRUE;
          break;
        }
        if (1 != sscanf (buf, "%i", &node))
          {
            printf ("%s: %s: buf = '%s'\n", PACKAGE, __FUNCTION__, buf);
            exit (1);
          }

        /* Create a mapping between internal node I and external NODE. */
        ais->node_map[node] = i++;
      }

  fclose (stream);

  /* Set the correct number of nodes.  This is later used to decide how many
     populations to generate. */
  ais->config->nodes = i;

  return;
}

/* Test the matching capabilities of the AIS with regard to speed and number
   of matches.  Mostly used for debugging purposes. */
void
ais_match_test (ais_t *ais)
{
  bitstring_match_test (ais->config->n_immature_max, ais->config->len,
                        ais->config->r);

  return;
}

/* Create a new AIS instance. */
ais_t *
ais_new (void)
{
  /* Allocate memory for a new ais_t instance. */
  ais_t *ais = (ais_t *) malloc (sizeof (*ais));

  /* Initialize all member variables. */
  ais->config = ais_config_new ();
  ais_config_initialize (ais->config);

  ais->current_file = 0;
  ais->node_list = NULL;

  return ais;
}

/* Clean up an ais_t instance.  I.e., free the memory used by member
   variables.  Performed in a separate function as it is a little bit more
   complex than simple free () calls. */
void
ais_clean_up (ais_t *ais)
{
  unsigned int n;

  for (n = 0; n < ais->config->nodes; n++)
    {
      node_destroy (&(ais->node_list[n]));
    }

  free (ais->node_list);

  /* Pop the allocator off the GSList allocator stack.            */
  g_slist_pop_allocator ();

  /* Free all the memory used by the allocator. */
  g_allocator_free (ais->allocator);

  return;
}

/* Destroy an ais_t instance. */
void
ais_destroy (ais_t *ais)
{
  /* Clean up the ais_t.      I.e. release, or free, more complex member
     variables. */
  ais_clean_up (ais);

  /* Destroy the config object. */
  ais_config_destroy (ais->config);

  /* Free memory allocated for this instance. */
  free (ais);

  return;
}

/* Converts an abstractly represented TCP network_connection_t into a
   bitstring_t.      This is done according to rules, partly described below and
   more thoroughly described elsewhere. */
void
ais_agent_from_tcp_connection (bitstring_t **agent, const unsigned int len,
                               const network_connection_t *connection)
{
  unsigned long long data = 0ULL;
  unsigned long long a = 0, b = 0, service = 0;
  unsigned long long server_flag = FALSE;
  unsigned int port = 0;

  /* In general, the following schema describes the rules for how the
     bitstring is created:
     Field   Bits   Description
     A        0–7   Internal host, or (if both hosts are internal) internal
     server.
     B       8–39   External host, or (if both hosts are internal) internal
     client.
     C         40   Server flag, set if the host in field A is server.
     D      41–48   Service identifier. */
  if ((connection->flags
       & (CF_INTERNAL_SOURCE | CF_SERVER_SOURCE))
      == (CF_INTERNAL_SOURCE | CF_SERVER_SOURCE))
    {
      a = connection->source;
      b = connection->destination;
      port = connection->source_port;
      server_flag = TRUE;
    }
  else if ((connection->flags
           & (CF_INTERNAL_DESTINATION | CF_SERVER_SOURCE))
          == (CF_INTERNAL_DESTINATION))
    {
      a = connection->destination;
      b = connection->source;
      port = connection->destination_port;
      server_flag = TRUE;
    }
  else if ((connection->flags
           & (CF_INTERNAL_SOURCE | CF_INTERNAL_DESTINATION
              | CF_SERVER_SOURCE))
          == (CF_INTERNAL_SOURCE))
    {
      a = connection->source;
      b = connection->destination;
      port = connection->destination_port;
      server_flag = FALSE;
    }
  else if ((connection->flags
           & (CF_INTERNAL_SOURCE | CF_INTERNAL_DESTINATION
              | CF_SERVER_SOURCE))
          == (CF_INTERNAL_DESTINATION | CF_SERVER_SOURCE))
    {
      a = connection->destination;
      b = connection->source;
      port = connection->source_port;
      server_flag = FALSE;
    }
  else
    {
      printf ("%s: %s: Where the shit hits the fan...\n",
              PACKAGE, __FUNCTION__);
      exit (1);
    }

  /* Map the server port number to a service identifier. */
  service = network_get_service (port);
  //printf ("port = %d, service = %qu\n", port, service);

  /* Store the information in the binary string. */
  data = ((service << 41)
         | (server_flag << 40)
         | (b << 9)
         | (a & 0x000000ff));

  /* Create AGENT, a peptide representing the network connection. */
  *agent = bitstring_new (len, &data);

  return;
}

/* Converts an abstractly represented network_connection_t into a bitstring_t.
```

```
   This is done according to rules, partly described below and more thoroughly
   described elsewhere. */
inline void
ais_agent_from_connection (bitstring_t **agent, const unsigned int len,
                           const network_connection_t *connection)
{
  unsigned long long data = 0ULL;

  /* In general, the following schema describes the rules for how the
     bitstring is created:
     Field    Bits   Description
     A        0–7    Internal host, or (if both hosts are internal) internal
     server.
     B        8–39   External host, or (if both hosts are internal) internal
     client.
     C          40   Server flag, set if the host in field A is server.
     D       41–48   Service identifier. */

  /* Store the information in the binary string. */
  data = (((unsigned long long)
           connection->destination_port << 41) /* service */
          | ((unsigned long long)
             connection->source_port << 40) /* local_is_server */
          | ((unsigned long long)
             connection->destination << 9) /* "remote_host" */
          | ((unsigned long long)
             connection->source & 0x000000ff)); /* "local_host" */

  /* Create AGENT, a peptide representing the network connection. */
  *agent = bitstring_new (len, &data);

  return;
}

/* Simulate the insertion of AGENT into the AIS. */
inline void
ais_insert_agent (ais_t *ais, const bitstring_t *agent)
{
  unsigned int n;

  /* Traverse the nodes. */
  for (n = 0; n < ais->config->nodes; n++)
    {
      /* Isn't touched every run if simulating a switched network. */
      ais->node_list[n].info.premature_deaths = 0;
      ais->node_list[n].info.false_negatives = 0;
      ais->node_list[n].info.true_negatives = 0;
      ais->node_list[n].info.false_positives = 0;
      ais->node_list[n].info.true_positives = 0;
      ais->node_list[n].info.activations = 0;
      ais->node_list[n].info.sig_1_only = 0;
      ais->node_list[n].info.sig_1_and_2 = 0;

      /* Insert the agent into every node. */
      node_insert_agent (&(ais->node_list[n]), agent);
    }
  return;
}

/* Generate ouput according to the state of the system.        The function is
    intended for generating data for statistical analysis. */
inline void
ais_generate_output (FILE *stream, const ais_t *ais,
                     const network_connection_t *connection)
{
  unsigned int n;
  //unsigned int detections = 0, premature_deaths = 0;
  //struct timeval now = {0, 0};

  /* Generate a timestamp. */
  //gettimeofday (&now, NULL);
  //timersub (&now, &ais->start, &now);

  /* Print the file sequence number so it should be easy to detect
      false-negative and false-positive results. */
  printf ("%i", ais->current_file);

  for (n = 0; n < ais->config->nodes; n++)
    {
      //detections += ais->node_list[n].info.detections;
      //premature_deaths += ais->node_list[n].info.premature_deaths;

      /* The total number of matches is not printed, as this number is equal
          to (PREMATURE_DEATHS + DETECTIONS) and thus may be calculated
          later. */
      printf (":%i %i %i %i %i %i %i %i",
              ais->node_list[n].info.premature_deaths,
              ais->node_list[n].info.false_negatives,
              ais->node_list[n].info.true_negatives,
              ais->node_list[n].info.false_positives,
              ais->node_list[n].info.true_positives,
              ais->node_list[n].info.activations,
              ais->node_list[n].info.sig_1_only,
              ais->node_list[n].info.sig_1_and_2);

      printf (" %i %i", ais->node_list[n].n_mature,
              ais->node_list[n].n_memory);
    }

  printf ("\n");

  //fprintf (stream, "%li.%06li %i %i %i %i\n",
  //         now.tv_sec, now.tv_usec, ais->current_file,
  //         matches, detections, premature_deaths);


  /* The total number of matches is not printed, as this number is equal to
      (PREMATURE_DEATHS + DETECTIONS) and thus may be calculated later. */
  //printf ("%i %i %i\n",
  //         ais->current_file,
  //         premature_deaths, detections);

  return;
```

```
}

/* Output a timestamp (NOW - START) to STREAM. */
inline void
ais_print_timestamp (FILE *stream, const struct timeval *start)
{
  struct timeval now;

  gettimeofday (&now, NULL);
  timersub (&now, start, &now);

  fprintf (stream, "# time = %li.%06li\n", now.tv_sec, now.tv_usec);

  return;
}

/* Run the artificial immune system in a simulated switched LAN. */
void
ais_run_switched (ais_t *ais)
{
  unsigned int f;
  network_next_connection_func_t next_connection = NULL;
  agent_from_connection_func_t agent_from_connection = NULL;
  FILE *stream = NULL;

  if (ais->config->file_format == AIS_FILE_FORMAT_TCP)
    {
      /* Initialize the network module (i.e., generate port mapping). */
      network_initialize ();

      /* Setup pointers to functions according to the file format. */
      next_connection = network_next_tcp_connection;
      agent_from_connection = ais_agent_from_tcp_connection;
    }
  else
    {
      /* Setup pointers to functions according to the file format. */
      next_connection = network_next_connection;
      agent_from_connection = ais_agent_from_connection;
    }

  /* Store a timestamp, recording when the program was started. */
  gettimeofday (&ais->start, NULL);

  for (f = 0; f < ais->config->file_names; f++)
    {
      unsigned int done = FALSE;
      ais->current_file = f;

      ais_print_timestamp (stdout, &ais->start);
      fprintf (stdout, "# Opening %s . . .\n", ais->config->file_name_list[f]);

      stream = network_open (ais->config->file_name_list[f]);

      while (! done)
        {
          network_connection_t connection;
```

```
            if (next_connection (stream, &connection))
              {
                done = TRUE;
                break;
              }
            else
              {
                bitstring_t *agent = NULL;

                /* Create an agent based on the connection data. */
                agent_from_connection (&agent, ais->config->len,
                                       &connection);

                /* Set the classification of the current agent.      Used for
                   precise statistics and simulation of manual costimulation. */
                ais->config->current_class = connection.class;

                /* Insert the AGENT into the involved node. */
                node_insert_agent (&ais->node_list
                                   [(int) ais->node_map[connection.source]],
                                   agent);

                /* If source_port (local_is_server) is set, then both source and
                   destination are internal hosts on the LAN. */
                if (connection.source_port == 1)
                  {
                    /* Insert the AGENT into the involved node. */
                    node_insert_agent (&ais->node_list
                                       [(int) ais->node_map
                                        [connection.destination & 0x000000ffU]],
                                       agent);
                  }

                /* Destroy AGENT after we've used it. */
                bitstring_destroy (agent);

                /* Output data describing the state of the system. */
                ais_generate_output (stdout, ais, &connection);
              }
          }

        fflush (stderr);

        fprintf (stdout, "# . . . closing %s\n", ais->config->file_name_list[f]);
        network_close (stream);
      }

  ais_print_timestamp (stdout, &ais->start);

  return;
}

/* Run the artificial immune system. */
void
ais_run_broadcast (ais_t *ais)
{
```

```
unsigned int f;
network_next_connection_func_t next_connection = NULL;
agent_from_connection_func_t agent_from_connection = NULL;
FILE *stream = NULL;

if (ais->config->file_format == AIS_FILE_FORMAT_TCP)
  {
    /* Initialize the network module (i.e., generate port mapping). */
    network_initialize ();

    /* Setup pointers to functions according to the file format. */
    next_connection = network_next_tcp_connection;
    agent_from_connection = ais_agent_from_tcp_connection;
  }
else
  {
    /* Setup pointers to functions according to the file format. */
    next_connection = network_next_connection;
    agent_from_connection = ais_agent_from_connection;
  }

/* Store a timestamp, recording when the program was started. */
gettimeofday (&ais->start, NULL);

for (f = 0; f < ais->config->file_names; f++)
  {
    unsigned int done = FALSE;
    ais->current_file = f;

    ais_print_timestamp (stdout, &ais->start);
    fprintf (stdout, "# Opening %s . . .\n", ais->config->file_name_list[f]);

    stream = network_open (ais->config->file_name_list[f]);

    while (! done)
      {
        network_connection_t connection;

        if (next_connection (stream, &connection))
          {
            done = TRUE;
            break;
          }
        else
          {
            bitstring_t *agent = NULL;

            /* Create an agent based on the connection data. */
            agent_from_connection (&agent, ais->config->len,
                                   &connection);

            /* Set the classification of the current agent.      Used for
               precise statistics and simulation of manual costimulation. */
            ais->config->current_class = connection.class;

            /* INSERT the AGENT into the AIS. */
            ais_insert_agent (ais, agent);
```

```
                /* Destroy AGENT after we've used it. */
                bitstring_destroy (agent);

                /* Output data describing the state of the system. */
                ais_generate_output (stdout, ais, &connection);
              }
          }

        fflush (stderr);

        fprintf (stdout, "# . . . closing %s\n", ais->config->file_name_list[f]);
        network_close (stream);
      }

  ais_print_timestamp (stdout, &ais->start);

  return;
}
```

A.2 AIS_CONFIG.C

```
/* $Id: ais_config.c,v 1.15 2002/05/27 18:44:44 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif /* HAVE_CONFIG_H */

#include "ais_config.h"

#include <stdio.h>
#include <stdlib.h>

#define AIS_DEFAULT_NODES 8
#define AIS_DEFAULT_POPULATION_SIZE 100
#define AIS_DEFAULT_LEN 49
#define AIS_DEFAULT_R 5
#define AIS_DEFAULT_DETECTOR_IMMATURE_PERIOD 350000
#define AIS_DEFAULT_DETECTOR_LIFETIME (5 * \
                                       AIS_DEFAULT_DETECTOR_IMMATURE_PERIOD)
#define AIS_DEFAULT_DETECTOR_ACTIVATION_LEVEL_MIN 1.0
#define AIS_DEFAULT_DETECTOR_ACTIVATION_LEVEL_MAX 2.0
#define AIS_DEFAULT_DETECTOR_COSTIMULATION_DELAY 2 //25000

/* The following parameters should assure fast increases and slower decay in
   the activation levels in the detectors. */
#define AIS_DEFAULT_DETECTOR_ACTIVATION_INC 1.5 /* Increase with 50%. */
#define AIS_DEFAULT_DETECTOR_ACTIVATION_DEC 0.8 /* Decrease with 20%. */

/* The following parameters are similar to the ones above, but are used for
   MEMORY detectors. */
#define AIS_DEFAULT_MEMORY_DETECTOR_ACTIVATION_INC 2.0 /* Increase with
```

```
                                                      100%. */
#define AIS_DEFAULT_MEMORY_DETECTOR_ACTIVATION_DEC 1.0 /* Decrease with 0%. */

#define AIS_DEFAULT_N_MATURE_MAX 20
#define AIS_DEFAULT_N_MEMORY_MAX 10

#define AIS_DEFAULT_NODE_IDS_FILENAME (NULL)
#define AIS_DEFAULT_FILE_FORMAT AIS_FILE_FORMAT_AIS

/* Default values concerned with somatic hypermutation. */
#define AIS_DEFAULT_SOMATIC_HYPERMUTATION 0
#define AIS_DEFAULT_P_MUTATE 0.5
#define AIS_DEFAULT_CLONES 15
#define AIS_DEFAULT_CLONES_SELECT 3

/* Default value to determine wheter we're going to simulate thymus
   behavior. */
#define AIS_DEFAULT_THYMUS 0

/* Default value to determine wheter we're going to simulate thymus
   behavior. */
#define AIS_DEFAULT_SOMATIC_HYPERMUTATION_COMPETITION 0

/* Create a new ais_config_t instance. */
ais_config_t *
ais_config_new (void)
{
  /* Allocate memory for an ais_config_t object and check that it was
     allocated successfully. */
  ais_config_t *config = (ais_config_t *) malloc (sizeof (*config));
  if (config == NULL)
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  return config;
}

/* Destroy an ais_config_t object */
void
ais_config_destroy (ais_config_t *config)
{
  /* Free memory allocated for member variables of this object. */
  free (config->file_name_list);

  /* Free memory allocated for the object itself. */
  free (config);

  return;
}

/* Initialize an ais_config_t instance. */
void
ais_config_initialize (ais_config_t *config)
{
  /* Set the default values for the behavior controlling parameters of AIS.
```

```
     Note that these values may be changed changed at runtime by command line
     options. */
  config->nodes = AIS_DEFAULT_NODES;
  config->n_immature_max = AIS_DEFAULT_POPULATION_SIZE;
  config->n_mature_max = AIS_DEFAULT_N_MATURE_MAX;
  config->n_memory_max = AIS_DEFAULT_N_MEMORY_MAX;

  config->len = AIS_DEFAULT_LEN;
  config->r = AIS_DEFAULT_R;

  config->match_test = FALSE;

  config->detector_lifetime = AIS_DEFAULT_DETECTOR_LIFETIME;
  config->detector_immature_period = AIS_DEFAULT_DETECTOR_IMMATURE_PERIOD;
  config->file_names = 0;
  config->file_name_list = NULL;

  config->detector_activation_level_min =
    AIS_DEFAULT_DETECTOR_ACTIVATION_LEVEL_MIN;
  config->detector_activation_level_max =
    AIS_DEFAULT_DETECTOR_ACTIVATION_LEVEL_MAX;
  config->detector_activation_inc =
    AIS_DEFAULT_DETECTOR_ACTIVATION_INC;
  config->detector_activation_dec =
    AIS_DEFAULT_DETECTOR_ACTIVATION_DEC;
  config->memory_detector_activation_inc =
    AIS_DEFAULT_MEMORY_DETECTOR_ACTIVATION_INC;
  config->memory_detector_activation_dec =
    AIS_DEFAULT_MEMORY_DETECTOR_ACTIVATION_DEC;
  config->costimulation_delay =
    AIS_DEFAULT_DETECTOR_COSTIMULATION_DELAY;

  config->file_format = AIS_DEFAULT_FILE_FORMAT;

  config->node_ids = AIS_DEFAULT_NODE_IDS_FILENAME;

  config->somatic_hypermutation = AIS_DEFAULT_SOMATIC_HYPERMUTATION;
  config->p_mutate = AIS_DEFAULT_P_MUTATE;
  config->clones = AIS_DEFAULT_CLONES;
  config->clones_select = AIS_DEFAULT_CLONES_SELECT;

  config->thymus = AIS_DEFAULT_THYMUS;

  config->sh_competition =
    AIS_DEFAULT_SOMATIC_HYPERMUTATION_COMPETITION;

  return;
}
```

A.3 AIS_CONFIG.H

```
/* $Id: ais_config.h,v 1.7 2002/05/27 18:44:45 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */
```

```
#ifndef AIS_CONFIG_H
#define AIS_CONFIG_H

#ifndef FALSE
#define FALSE 0                    /* This is the naked Truth. */
#endif /* ! FALSE. */

#ifndef TRUE
#define TRUE 1                     /* ... and this is the Light. */
#endif /* ! TRUE. */

#include "network.h"

typedef enum
{
  AIS_FILE_FORMAT_NONE = 0x00,
  AIS_FILE_FORMAT_AIS = 0x01,
  AIS_FILE_FORMAT_TCP = 0x01 << 1
}
ais_file_format_t;

/* A data structure comprising the configuration information for AIS. */
struct _ais_config
{
  /* The number of nodes in the AIS. */
  unsigned int nodes;

  /* The populations size at each node. */
  unsigned int n_immature_max;

  /* Maximum number of MATURE detectors per node. */
  unsigned int n_mature_max;

  /* Maximum number of MEMORY detectors per node. */
  unsigned int n_memory_max;

  /* The length of each bitstring. */
  unsigned int len;

  /* The r of the r-contiguous matching algorightm. */
  unsigned int r;

  /* Whether the system should perform a benchmark test of the matching
     algorithm. */
  unsigned int match_test;

  /* The lenght of a detectors lifetime. */
  unsigned int detector_lifetime;

  /* The length of a detectors childhood.        Must be shorter than
     DETECTOR_LIFETIME. */
  unsigned int detector_immature_period;

  /* The lowest level of activation for a detector. */
  double detector_activation_level_min;
```

```
/* The activation threshold.     If the activation threshold exceeds this, the
   state of the detector changes to DS_ACTIVE. */
double detector_activation_level_max;

/* A factor that the activation level is muliplied with to increase it. */
double detector_activation_inc;

/* A factor that the activation level is muliplied with to decrease it. */
double detector_activation_dec;

/* A factor that the activation level is muliplied with to increase it for
   MEMORY detectors. */
double memory_detector_activation_inc;

/* A factor that the activation level is muliplied with to decrease it for
   MEMORY detectors. */
double memory_detector_activation_dec;

/* The time a detector has to wait for its costimulation signal. */
unsigned int costimulation_delay;

/* The number of file names to be included in the simulation. */
unsigned int file_names;

/* An array of file names.     Should be FILE_NAMES elements long. */
unsigned char **file_name_list;

/* A symbol representing the file format of the input files.          May be either
   TCP or AIS. */
ais_file_format_t file_format;

/* A filename.     The file contains a (possible unordered) list of all the
   possible internal (i.e., on the LAN) nodes in the test data. */
unsigned char *node_ids;

/* A TRUE or FALSE flag to determine if somatic hypermutation should be
   performed. */
unsigned int somatic_hypermutation;

/* A TURE or FALSE flag indicating if "thymus" behavior should be
   simulated. */
unsigned int thymus;

/* A TRUE or FALSE flag indicating if the new somatic hypermutation
   behavior should be simulated. */
unsigned int sh_competition;

/* Determines the probability of a mutation occuring under somatic
   hypermutation. */
double p_mutate;

/* Number of clones to generate in the somatic hypermutation process. */
unsigned int clones;

/* Number of clones to use after somatic hypermutation. */
unsigned int clones_select;
```

```
  /* Classification of current agent (self, nonself or unknown). */
  connection_class_t current_class;
};
typedef struct _ais_config ais_config_t;

ais_config_t *ais_config_new (void);

void ais_config_destroy (ais_config_t *config);

void ais_config_initialize (ais_config_t *config);

#endif /* AIS_CONFIG_H */
```

A.4 BITSTRING.C

```
/* $Id: bitstring.c,v 1.24 2002/05/30 21:39:16 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif /* HAVE_CONFIG_H */

#include "bitstring.h"
#include "mt19937ar_cok.h"

#ifdef HAVE_FCNTL_H
#include <fcntl.h>
#endif /* HAVE_FCNTL_H */

#ifdef HAVE_LIMITS_H
#include <limits.h>
#endif /* HAVE_LIMITS_H */

#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif /* HAVE_UNISTD_H */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* The number of bits in an unsigned long long. */
#define ULLONG_BITS (sizeof (unsigned long long) * CHAR_BIT)

/* Define some constants to speed up division and mod () operations where the
   operator is defined a powers of two. */
#define DIV_8 3              /* (x / 8) == (x >> 3).   8 = (1 << 3). */
#define MOD_8 (8 - 1)        /* (x % 8) == (x & MOD_8). */
#define DIV_32 5             /* (x / 32) == (x >> 5).   32 = (1 << 5). */
#define MOD_32 (32 - 1)      /* (x % 32) == (x & MOD_32). */
#define DIV_64 6             /* (x / 64) == (x >> 6).   64 = (1 << 6). */
#define MOD_64 (64 - 1)      /* (x % 64) == (x & MOD_64). */

```

```
/* The device to be used as a random generator.   This device gathers
   environmental noise from device drivers, etc., and returns good random
   numbers, suitable for cryptographic use. */
#define RANDOM_DEVICE "/dev/urandom"

#if defined __GNUC__ && __GNUC__ >= 2

/* A macro function definition, to be used inline where it is called.     The
   function makes use of special assembly statements, available on Intel
   Pentium architectures (and higher) to check wether bit B in D is set. */
#define bit_isset(d, b) \
  (__extension__ \
   ({ \
     register char __result; \
     __asm__ __volatile__ \
     ("btl %1, %2 ; setcb %b0" \
                          : "=q" (__result) \
                          : "r" (b), "m" (d) \
                          : "cc"); \
     __result; \
   }))

#endif /* __GNUC__ && __GNUC__ >= 2 */

#ifndef max
#define max(X, Y) \
     ({ typeof (X) x_ = (X); \
     typeof (Y) y_ = (Y); \
     (x_ > y_) ? x_ : y_; })
#endif /* ! max. */

#ifndef min
#define min(X, Y) \
     ({ typeof (X) x_ = (X); \
     typeof (Y) y_ = (Y); \
     (x_ < y_) ? x_ : y_; })
#endif /* ! min. */

/* Intialize the MT19937 pseudorandom number generator. */
void
random_generator_init (void)
{
  unsigned long seed = 0UL;
  size_t size = sizeof (seed);
  ssize_t len = 0;
  /* Open the RANDOM_DEVICE. */
  int fd = open (RANDOM_DEVICE, O_RDONLY);

  if (fd == −1)
    {
      perror (PACKAGE ": " __FUNCTION__);
      exit (1);
    }

  /* Retrieve a 32 bit seed from RANDOM_DEVICE. */
  while ((unsigned int) len < size)
    {
        /* Read (size - len) number of bytes into the memory starting at address
           (buf + len). */
        ssize_t len_d = read (fd, (&seed + len), (size - len));

        /* If the number of bytes actually read is negative, a serious error has
           occured. */
        if (len_d < 0)
          {
            perror (__FUNCTION__ ": read ()");
            exit (1);
          }

        /* Add the number of bytes actually read to LEN. */
        len += len_d;
      }

    /* Close the RANDOM_DEVICE. */
    close (fd);

    /* Seed the Mersenne Twister 19937 pseudorandom number generator. */
    mt19937_init_genrand (seed);

    return;
  }

  /* Retrieve BITS random bits store the result in BUF.   Requires that both
     BUF >= 32 bits and BITS > 0. */
  void
  random_generator_randbits (const unsigned int bits, void *buf)
  {
    unsigned int units = 1 + ((bits - 1) >> DIV_32); /* Divide by 32. */
    unsigned int n = 0;
    unsigned int b = (bits & MOD_32);

    while (n < units)
      {
        ((unsigned long *) buf)[n] = random_generator_rand_int32 ();
        n++;
      }

    if (b > 0)
      {
        /* Mask out the excess bits (that should not be used).   Set them to 0.*/
        ((unsigned long *) buf)[n - 1] &= (0xffffffffUL >> (32 - b));
      }

    return;
  }

  /* Short-cuts the 'bitstring_destroy () -> bitstring_new ()' sequence and
     returns a recycled bitstring_t instance, without freeing and reallocating
     the memory for the data member variable.   Should be used when the size of
     the instance is not changed.   Behaves very much like bitstring_new (), see
     below. */
  inline void
  bitstring_recycle (bitstring_t *bitstring, const unsigned int size,
                     const unsigned long long *data)
```

```
{
  unsigned int units = 1 + ((size - 1) >> DIV_64);

  bitstring->size = size;

  if (data == NULL)
    {
      /* If no data pointer was provided, set all the bits in the bitstring to
         0. */
      memset (bitstring->data, 0, (sizeof (*bitstring->data) * units));
    }
  else
    {
      /* Copy all the data from DATA to bitstring->data. */
      while (units)
        {
          bitstring->data[units - 1] = data[units - 1];
          units -= 1;
        }
    }

  return;
}

/* Create a new bitstring_t incstance and allocates memory for its data member
   variables.   Requires the size (in the number of bits) and a pointer to the
   data to store as arguments.   The data are copied. */
inline bitstring_t *
bitstring_new (const unsigned int size, const unsigned long long *data)
{
  const unsigned int units = 1 + ((size - 1) >> DIV_64); /* Divide by 64. */
  bitstring_t *new = (bitstring_t *) malloc (sizeof (*new));
  if (new == NULL)
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  new->data = (unsigned long long *) malloc (sizeof (*(new->data)) * units);
  if (new->data == NULL)
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  bitstring_recycle (new, size, data);

  return new;
}

/* Destroy a bitstring_t instance by freeing the allocated memory in the
   reverse order of which it was allocated. */
inline void
bitstring_destroy (bitstring_t *bitstring)
{
  free (bitstring->data);
  free (bitstring);
  return;
}

/* Output BITSTRING as a binary number in ASCII format. */
inline void
bitstring_print (bitstring_t *bitstring)
{
  unsigned int i = bitstring->size;

  while (i)
    {
      printf (bit_isset (*bitstring->data, (i - 1)) ? "1" : "0");
      i -= 1;
    }

  return;
}

/* Create a new bitstring_t instance with SIZE number of random bits,
   retrieved from RANDOM_GENERATOR. */
bitstring_t *
bitstring_create_random (const unsigned int size)
{
  unsigned long long data[(1 + ((size - 1) >> DIV_64))];

  random_generator_randbits (size, data);

  return bitstring_new (size, data);
}

/* The bitstring_recycle_random () differs to bitstring_create_random () just
   as the bitstring_recycle () differs to bitstring_new ().      Should be used
   where applicable to increase performance. */
inline void
bitstring_recycle_random (bitstring_t *bitstring, const unsigned int size)
{
  bitstring_recycle (bitstring, size, NULL);

  random_generator_randbits (size, bitstring->data);

  return;
}

/* Perform a bitwise XOR on two bitstring_t instances A and B and
   return the result in the newly allocated NEW.      The result instance
   must be freed after use. */
bitstring_t *
bitstring_xor (const bitstring_t *a, const bitstring_t *b)
{
  bitstring_t *new = bitstring_new (a->size, a->data);
  unsigned int i = 1 + ((new->size - 1) >> DIV_64);

  while (i)
    {
      new->data[i - 1] = a->data[i - 1] ^ b->data[i - 1];
      i -= 1;
    }

  return new;
}

/* Return a boolean value indicating whether there are R contiguous,
     pairwise equal, bits present in the bitstrings A and B. */
inline unsigned int
bitstring_r_contiguous (const bitstring_t *a, const bitstring_t *b,
                        const unsigned int r)
{
  register unsigned int c, i;
  register unsigned int bits = a->size;
  const unsigned int units = 1 + ((bits - 1) >> DIV_64);

  if (! r)
    {
      return 1;
    }

  for (c = 0, i = 0; i < units; i++, bits -= ULLONG_BITS)
    {
      register unsigned long long xor;
      register unsigned int j;

      /* Perform a bitwise XOR on the two units at position I, and
           store the result in XOR, where all bits that is pairwise
           unequal i A and B are set to 1, and 0 otherwise. */
      xor = (*(a->data + i) ^ *(b->data + i));

      /* For this unit, bitshift through all the remaining bits and
           count the number of contiguous false bits. */
      for (j = 0; j < bits; j++)
        {
          if (xor & (1ULL << j))
            {
              /* Start counting from 0. */
              c = 0;
            }
          else
            {
              /* Add 1 to the count of contiguous false bits. */
              c += 1;

              if (r == c)
                {
                  /* The R constraint was met, report a positive
                       test. */
                  return 1;
                }
            }
        }
    }

  return 0;
}
```

```
/* Return a value indicating the maximum R contiguously affinity
   between A and B.  The AFFINITY value is the highest number of
   contiguous bits occuring in A and B. */
inline unsigned int
bitstring_contiguous_affinity (const bitstring_t *a, const bitstring_t *b)
{
  register unsigned int c, i;
  register unsigned int bits = a->size;
  register unsigned int affinity = 0;
  const unsigned int units = 1 + ((bits - 1) >> DIV_64);

  for (c = 0, i = 0; i < units; i++, bits -= ULLONG_BITS)
    {
      register unsigned long long xor;
      register unsigned int j;

      /* Perform a bitwise XOR on the two units at position I, and
         store the result in XOR, where all bits that is pairwise
         unequal i A and B are set to 1, and 0 otherwise. */
      xor = (*(a->data + i) ^ *(b->data + i));

      /* For this unit, bitshift through all the remaining bits and
         count the number of contiguous false bits. */
      for (j = 0; j < bits; j++)
        {
          if (xor & (1ULL << j))
            {
              /* Start counting from 0. */
              c = 0;
            }
          else
            {
              /* Add 1 to the count of contiguous false bits. */
              c += 1;

              /* Set or increase AFFINITY. */
              affinity = max (affinity, c);
            }
        }
    }

  return affinity;
}

/* Perform a benchmarking test to measure the performance of the
   bitstring_r_contiguous test.    To generate performance statistics,
   compile with the flag '-pg' and use gprof. */
void
bitstring_match_test (const unsigned int pop_size,
                      const unsigned int len,
                      const unsigned int r)
{
  unsigned int lenz = len /* 49 */;
  bitstring_t *d = bitstring_create_random (lenz);
  bitstring_t *t = bitstring_create_random (lenz);
  register unsigned long long m = 0;
  register unsigned long long i;
  unsigned int size = 1000000000;

  for (i = 0; i < size; i++)
    {
      bitstring_recycle_random (t, lenz);

      if (bitstring_r_contiguous (d, t, r))
        {
          m++;
        }
    }

  bitstring_destroy (t);
  bitstring_destroy (d);

  printf ("Matches: %g * 2^%d => %qu (of %d)\n",
          (m / (double) size), lenz,
          (unsigned long long)
          ((m / (double) size) * (1ULL << lenz)), (size));

  return;
}

/* Test the behavior of the bitstring module. */
void
bitstring_test (void)
{
  const unsigned int pop_size = 100;
  const unsigned int len = 49;
  const unsigned int r = 5;
  bitstring_t *a, *b;
  unsigned int n;

  random_generator_init ();

  for (n = 0; n < 1; n++)
    {
      bitstring_t *c;
      unsigned int i;

      a = bitstring_create_random (len);
      b = bitstring_create_random (len);
      c = bitstring_xor (a, b);
      i = bitstring_r_contiguous (a, b, r);

      printf ("a: "); bitstring_print (a); puts ("");
      printf ("b: "); bitstring_print (b); puts ("");
      printf ("c: "); bitstring_print (c); puts ("");
      printf ("t: %d\n", i);

      bitstring_destroy (c);
      bitstring_destroy (b);
      bitstring_destroy (a);
    }

  bitstring_match_test (pop_size, len, r);
  return;
}
```

A.5 BITSTRING.H

```
/* $Id: bitstring.h,v 1.12 2002/05/27 18:44:45 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */

#ifndef BITSTRING_H
#define BITSTRING_H

/*! The bitstring structure. */
struct _bitstring
{
  unsigned long long *data;     /*! The data representing the bitstring. */
  unsigned int size;            /*! Number of bits used for a bitstring. */
};
typedef struct _bitstring bitstring_t;

#include "mt19937ar_cok.h" /* Mersenne Twister 19937 implementation. */

/* Initialize the Mersenne Twister (MT) random generator. */
void random_generator_init (void);

/* Generates unsigned 32-bit integers. */
#define random_generator_rand_int32() mt19937_genrand_int32()

/* Generates unsigned 31-bit integers. */
#define random_generator_rand_int31() mt19937_genrand_int31()

/* Generates uniform real in [0,1) (32-bit resolution). */
#define random_generator_rand_real() mt19937_genrand_real2()

/* Generates uniform real in [0,1] (32-bit resolution). */
#define random_generator_rand_real_closed() mt19937_genrand_real1()

/* Retrieves BITS random bits from FD and stores the result in BUF. */
void random_generator_randbits (const unsigned int bits,
                                void *buf);

/* Short-cuts the 'bitstring_destroy () -> bitstring_new ()' sequence and
   returns a recycled bitstring_t instance, without freeing and reallocating
   the memory for the data member variable. */
void bitstring_recycle (bitstring_t *bitstring, const unsigned int size,
                        const unsigned long long *data);

/* Create a new bitstring_t incstance and allocates memory for its data member
   variable.   SIZE is the length of the bitstring (in bits) and DATA is the
   initial data. */
bitstring_t *bitstring_new (const unsigned int size,
                            const unsigned long long *data);

/* Destroy a bitstring_t instance by freeing the allocated memory in the
```

```
   reverse order of which it was allocated. */
void bitstring_destroy (bitstring_t *bitstring);

/* Output BITSTRING as a binary number in ASCII format. */
void bitstring_print (bitstring_t *bitstring);

/* Create a new bitstring_t instance with SIZE number of random bits,
   retrieved from RANDOM_GENERATOR. */
bitstring_t *bitstring_create_random (const unsigned int size
                                      /*, const int random_generator*/);

/* The bitstring_recycle_random () differs to bitstring_create_random () just
   as the bitstring_recycle () differs to bitstring_new ().      Should be used
   where applicable to increase performance. */
void bitstring_recycle_random (bitstring_t *bitstring,
                               const unsigned int size
                               /*, const int random_generator*/);

/* Perform a bitwise XOR on two bitstring_t instances A and B and return the
   result in the newly allocated NEW.     The result instance must be freed after
   use. */
bitstring_t *bitstring_xor (const bitstring_t *a,
                            const bitstring_t *b);

/* Return a boolean value indicating whether there are R contiguous, pairwise
   equal, bits present in the bitstrings A and B. */
unsigned int bitstring_r_contiguous (const bitstring_t *a,
                                     const bitstring_t *b,
                                     const unsigned int r);

/* Return a value indicating the maximum R contiguously affinity
   between A and B.  The AFFINITY value is the highest number of
   contiguous bits occuring in A and B. */
unsigned int bitstring_contiguous_affinity (const bitstring_t *a,
                                            const bitstring_t *b);

/* Perform a benchmarking test to measure the performance of the
   bitstring_r_contiguous test.      To generate performance statistics, compile
   with the flag '-pg' and use gprof. */
void bitstring_match_test (const unsigned int pop_size,
                           const unsigned int len,
                           const unsigned int r
                           /*, const int fd*/);

/* Test the behavior of the bitstring module. */
void bitstring_test (void);

#endif /* ! BITSTRING_H */
```

A.6 DETECTOR.C

```
/* $Id: detector.c,v 1.17 2002/05/27 18:44:45 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */
```

```
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif /* HAVE_CONFIG_H */

#include <stdio.h>
#include <stdlib.h>

#include "detector.h"

#define BITSPERBYTE 8
#define BITS(type) (BITSPERBYTE * (int)sizeof(type))
#define LONGBITS BITS(long)

#define DIV_64 6                /* (x / 64) == (x >> 6).   64 = (1 << 6). */
#define MOD_64 (64 − 1)         /* (x % 64) == (x & MOD_64). */

#ifndef max
#define max(X, Y) \
     ({ typeof (X) x_ = (X); \
     typeof (Y) y_ = (Y); \
     (x_ > y_) ? x_ : y_; })
#endif /* ! max. */

#ifndef min
#define min(X, Y) \
     ({ typeof (X) x_ = (X); \
     typeof (Y) y_ = (Y); \
     (x_ < y_) ? x_ : y_; })
#endif /* ! min. */

#define ADDR (*(volatile long *) addr)
static __inline__ void bit_toggle(volatile void * addr, int nr)
{
  __asm__ __volatile__(
                       "btcl %1,%0"
                       :"=m" (ADDR)
                       :"Ir" (nr));
}

/* Initialize DETECTOR, a detector_t instance. */
inline void
detector_initialize (detector_t *detector,
                     const detector_state_t state,
                     const double activation_level_min)
{
  detector−>age = 0;
  detector−>activation = activation_level_min;
  detector−>state = state;
  detector−>activation_age = −1;
  detector−>antigen = NULL;

  return;
}

inline detector_t *
detector_new (const unsigned int size,
               const unsigned long long *data,
               const detector_state_t state,
               const double activation_level_min)
{
  detector_t *detector = (detector_t *) malloc (sizeof (*detector));
  if (detector == NULL)
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  detector->bitstring = bitstring_new (size, data);

  detector_initialize (detector, state, activation_level_min);

  return detector;
}

/* Unbind ANTIGEN from DETECTOR. */
inline void
detector_antigen_unbind (detector_t *detector)
{
  /* Destroy the ANTIGEN. */
  bitstring_destroy (detector->antigen);

  /* Be shure to indicate that DETECTOR is not bound to any
     ANTIGEN. */
  detector->antigen = NULL;

  return;
}

/* Bind ANTIGEN to DETECTOR. */
inline void
detector_antigen_bind (detector_t *detector, const bitstring_t *antigen)
{
  /* If DETECTOR is already bound to an antigen, unbind it. */
  if (detector->antigen != NULL)
    {
      detector_antigen_unbind (detector);
    }

  /* Create a copy of the antigen.  Remember that this must be freed
     (most probably done via the detector_atnigen_unbind () call. */
  detector->antigen = bitstring_new (antigen->size, antigen->data);

  return;
}

/* Recycle the memory allocated for DETECTOR to produce a _new_
   detector.  If DETECTOR is bound to any ANTIGEN, break up the
   binding (i.e., destroy ANTIGEN and free memory allocated for it).
   Also, its BITSTRING is recycled in the same mannor. */
inline void
detector_recycle (detector_t *detector, const unsigned int size,
                  const unsigned long long *data,
                  const detector_state_t state,
                         const double activation_level_min)
{
  /* If DETECTOR is bound to some ANTIGEN, unbind it. */
  if (detector->antigen != NULL)
    {
      detector_antigen_unbind (detector);
    }

  bitstring_recycle (detector->bitstring, size, data);
  detector_initialize (detector, state, activation_level_min);

  return;
}

/* Create a new detector with a randomly generated bitstring. */
detector_t *
detector_create_random (const unsigned int size,
                        const detector_state_t state,
                        const double activation_level_min)
{
  unsigned long long data[(1 + ((size - 1) >> DIV_64))];

  random_generator_randbits (size, data);

  return detector_new (size, data, state, activation_level_min);
}

/* Mutate DETECTOR with a probability P_MUTATE of a point mutation.
   If a mutation takes place, it is performed by toggling a randomly
   chosen bit I in the bitstring.   If P == 0.0, _no_ mutation takes
   place, while if P == 1.0, there certainly will be a mutation.
   Returns 0 if no mutation took place and 1 if it did. */
inline unsigned int
detector_mutate (detector_t *detector, const double p_mutate)
{
  /* Se if P_MUTATE is greater than a pseudorandomly generated real in
     the range [0, 1) with 32bit precision.   If P_MUTATE == 1.0 the
     conditional will always evaluate to true, and if P_MUTATE == 0.0
     the expression will always evaluate to false. */
  if (p_mutate > random_generator_rand_real ())
    {
      /* Get a random index (integer) in the range [0,
         detector->size). */
      unsigned int i = detector->bitstring->size
        * random_generator_rand_real ();

      /* Change the bit at the Ith position. */
      bit_toggle (&detector->bitstring->data[i >> DIV_64], (i & MOD_64));

      /* Increase the number of mutations.   This variable may be used
         for statistics and debugging. */
      return 1;
    }

  return 0;
}
```

```
/* Clone DETECTOR and return the new clone. */
inline detector_t *
detector_clone (const detector_t *detector)
{
  /* Create a copy of the current DETECTOR.  The new bitstring should
     be _exactly_ the same as the original. */
  detector_t *new = detector_new (detector->bitstring->size,
                                  detector->bitstring->data,
                                  detector->state, detector->activation);

  /* If DETECTOR is bound to some ANTIGEN, make shure NEW also binds
     to the same ANTIGEN. */
  if (detector->antigen != NULL)
    {
      detector_antigen_bind (new, detector->antigen);
    }

  return new;
}

/* Clone DETECTOR without allocating any new memory (i.e., use the
   memory allocated for CLONE). */
inline void
detector_recycle_clone (detector_t *clone, const detector_t *detector)
{
  /* Create a copy of the current DETECTOR.  The new bitstring should
     be _exactly_ the same as the original. */
  detector_recycle (clone, detector->bitstring->size,
                    detector->bitstring->data, detector->state,
                    detector->activation);

  /* If DETECTOR is bound to some ANTIGEN, make shure CLONE also binds
     to the same ANTIGEN. */
  if (detector->antigen != NULL)
    {
      detector_antigen_bind (clone, detector->antigen);
    }

  return;
}

/* Recycle DETECTOR by reusing the memory it occupies and simply
   storing a new randomly generated value into the bitstring data. */
inline void
detector_recycle_random (detector_t *detector, const unsigned int size,
                         const detector_state_t state,
                         const double activation_level_min)
{
  /* Note that if DETECTOR is bound to any antibody, this binding will
     be destroyed, and the antibody will be freed. */
  detector_recycle (detector, size, NULL, state, activation_level_min);

  random_generator_randbits (size, detector->bitstring->data);

  return;
}
```

```
/* Destroy the detector. */
inline void
detector_destroy (detector_t *detector)
{
  /* If DETECTOR is already bound to an ANTIGEN, unbind it. */
  if (detector->antigen != NULL)
    {
      detector_antigen_unbind (detector);
    }

  /* Call the destructor of the bitstring member. */
  bitstring_destroy (detector->bitstring);

  /* Free allocated memory. */
  free (detector);

  return;
}

/* Wrapper call to be used with g_list_foreach ().          DATA is not
   used. */
inline void
detector_destroy_fe_wrapper (detector_t *detector, void *data)
{
  detector_destroy (detector);
  return;
}

/* Increase the activation level of the detector.          If ACTIVATION has
   reached the threshold ACTIVATION_LEVEL_MAX, set the ds_active
   flag. */
inline void
detector_activation_increase (detector_t *detector,
                              const double activation_inc,
                              const double activation_level_max)
{
  detector->activation *= activation_inc;

  if (detector->activation >= activation_level_max)
    {
      detector->state |= DS_ACTIVE;
    }

  return;
}

/* Decrease the activation level of the detector.          The activation
   level is not allowed to get lower than ACTIVATION_LEVEL_MIN. */
inline void
detector_activation_decrease (detector_t *detector,
                              const double activation_dec,
                              const double activation_level_min)
{
  detector->activation *= activation_dec;

  if (detector->activation < activation_level_min)
    {
        detector->activation = activation_level_min;
      }

    return;
}

/* Increase the age of the detector.         Kind of ''Make another day pass
    by''. */
inline void
detector_age (detector_t *detector, const unsigned int immature_period,
              const unsigned int lifetime)
{
  /* Increase the age of the detector. */
  detector->age++;

  if (detector->age > lifetime)
    {
      if ((detector->state & DS_MEMORY) == DS_MEMORY)
        {
          /* Let memory detectors live for ever, avoid age
              wraparound. */
          detector->age = lifetime;
        }
      else
        {
          /* The detector has reached its end station. */
          detector->state |= DS_DEAD;
        }
    }
  else {
    if (detector->age > immature_period)
      {
        /* The detector has reached a mature (and naive) age. */
        detector->state |= DS_MATURE;
      }
  }

  return;
}
```

A.7 DETECTOR.H

```
/* $Id: detector.h,v 1.9 2002/05/27 18:44:45 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */

#ifndef DETECTOR_H
#define DETECTOR_H

#include "bitstring.h"

typedef enum
{
  DS_NONE = 0,
    DS_MATURE = 0x01 << 0,
    DS_ACTIVE = 0x01 << 1,
    DS_MEMORY = 0x01 << 2,
    DS_DEAD = 0x01 << 3,
    DS_COSTIMULATION_SELF = 0x01 << 4,
    DS_COSTIMULATION_NONSELF = 0x01 << 5
}
detector_state_t;

struct _detector
{
    bitstring_t *bitstring;         /* The antigen recognition site. */
    double activation;              /* Activation level of the
                                       detector. */
    detector_state_t state;         /* The state of the detector. */
    unsigned int age;               /* Age of the detector. */
    int activation_age;             /* When (AGE - COSTIMULATION_DELAY) ==
                                       ACTIVATION_AGE, the detector
                                       receives the costimulation
                                       signal. */
    bitstring_t *antigen;           /* If detector gets activated, this
                                       will be the antigen binding to
                                       it. */
};
typedef struct _detector detector_t;

detector_t *detector_new (const unsigned int size,
                          const unsigned long long *data,
                          const detector_state_t state,
                          const double activation_level_min);

detector_t *detector_create_random (const unsigned int size,
                                    const detector_state_t state,
                                    const double activation_level_min);


/* With a probability P_MUTATE, let DETECTOR undergo a point mutation.
   If a mutation takes place, it is performed by toggling a randomly
   chosen bit I in the bitstring.     If P == 0.0, _no_ mutation takes
   place, while if P == 1.0, there certainly will be a mutation.
   Returns 0 if no mutation took place and 1 if it did. */
unsigned int detector_mutate (detector_t *detector,
                              const double p_mutate);

/* Clone DETECTOR and return the new clone. */
detector_t *detector_clone (const detector_t *detector);

void detector_recycle_clone (detector_t *clone, const detector_t *detector);

void detector_recycle_random (detector_t *detector,
                              const unsigned int size,
                              const detector_state_t state,
                              const double activation_level_min);

void detector_destroy (detector_t *detector);

/* Unbind ANTIGEN from DETECTOR. */
```

```
void detector_antigen_unbind (detector_t *detector);

/* Bind ANTIGEN to DETECTOR. */
void detector_antigen_bind (detector_t *detector,
                            const bitstring_t *antigen);

/* Wrapper call to be used with g_list_foreach ().    DATA is not
   used. */
void detector_destroy_fe_wrapper (detector_t *detector, void *data);

void detector_activation_increase (detector_t *detector,
                                   const double activation_inc,
                                   const double activation_level_max);

void detector_activation_decrease (detector_t *detector,
                                   const double activation_dec,
                                   const double activation_level_min);

void detector_age (detector_t *detector, const unsigned int immature_period,
                   const unsigned int lifetime);

#endif /* ! DETECTOR_H */
```

A.8 NETWORK.C

```
/* $Id: network.c,v 1.10 2002/05/09 14:37:50 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif /* HAVE_CONFIG_H */

#include <stdlib.h>
#include <string.h>

#ifdef HAVE_FCNTL_H
#include <fcntl.h>
#endif /* HAVE_FCNTL_H */

#ifdef HAVE_UNISTD_H
#include <unistd.h>
#endif /* HAVE_UNISTD_H */

#include "network.h"

#define NETWORK_READ_BUF_SIZE 1024

/* Well-known services, based on FreeBSD 4.3's '/etc/services' and Steven
   Andrew Hofmeyr's Ph.D. dissertation (May 1999, University of New
   Mexico). */
static unsigned int services_well_known[] =
{
  /* Commonly assigned privileged ports. */
  1, 7, 9, 11, 13, 19, 20, 21, 22, 23, 25, 37, 38, 42, 43, 53, 68, 70, 79, 80,
  87, 94, 95, 109, 110, 111, 113, 119, 123, 130, 131, 132, 137, 138, 139, 143,
  156, 161, 162, 177, 178, 194, 199, 200, 201, 202, 203, 204, 205, 206, 207,
  208, 210, 213, 220, 372, 387, 396, 411, 443, 512, 513, 514, 515, 523, 540,
  566
};

/* A conversion table for the services mentioned above.  This enables a
   service number (port) to be encoded as a byte value, as the size of the
   above table is smaller than 256 (214). */
unsigned char network_service_table[NETWORK_WELL_KNOWN_SERVICES] =
{
  0,
};

/* An enumeration used to passed to parse_host_str () to indicate whether it
   is parsing a source or a destination address. */
typedef enum
{
  PHF_NONE = 0x00,
  PHF_SOURCE = 0x01 << 0,
  PHF_DESTINATION = 0x01 << 1
}
parse_host_flag_t;

/* Create a mapping table, so that the service of a connection can be
   normalized to be a value between 0 and 255. */
void
network_initialize (void)
{
  unsigned int i;
  unsigned int services = (sizeof (services_well_known) /
                           sizeof (*services_well_known));

  for (i = 0; i < services; i++)
    {
      network_service_table[services_well_known[i]] = (i + 1);
    }

  return;
}

/* Return the 'normalized' service, given PORT as input. */
unsigned char
network_get_service (const unsigned int port)
{
  if (port < NETWORK_WELL_KNOWN_SERVICES)
    {
      unsigned char s = network_service_table[port];

      if (s == 0)
        {
          /* The service uses a privileged port, but is not in the mapping
             table. */
          return (sizeof (services_well_known)
                  / sizeof (*services_well_known)) + 1;
        }
      return s;
    }
  else
    {
      /* Non-privileged port.   Return a separate value for this service. */
      return ((sizeof (services_well_known)
              / sizeof (*services_well_known)) + 2);
    }
}

FILE *
network_open (const char *path)
{
  FILE *stream = fopen (path, "r");
  if (stream == NULL)
    {
      perror (PACKAGE ": "__FUNCTION__);
      exit (1);
    }

  return stream;
}

int
network_close (FILE *stream)
{
  return fclose (stream);
}

inline double
time_str_to_numeral (const char *str)
{
  unsigned int hours = 0, minutes = 0;
  double seconds = 0.0, time = 0.0;

  if (3 == sscanf (str, "%u:%u:%lf", &hours, &minutes, &seconds))
    {
      time = ((double) (hours * 3600) + (double) (minutes * 60) + seconds);
    }
  else
    {
      perror (PACKAGE ": " __FUNCTION__);
      exit (1);
    }
  return time;
}

/* Convert the IP version 4 (IPv4) address and port number string str to a
   pair of unsigned int variables.   This requires IPv4, as these addresses may
   be represented by 32 bits. */
inline int
network_parse_host_str (const char *str, unsigned int *host_addr,
                        unsigned int *port, unsigned int *flags,
                        const parse_host_flag_t parse_flags)
{
  unsigned int h3, h2, h1, h0, p;
  switch (sscanf (str, "%u.%u.%u.%u.%u", &h0, &h1, &h2, &h3, &p))
    {
    case 5:
      *port = p;
      /* FALL THROUGH! */
    case 4:
      h2 = (h2 > 255) ? 255 : h2;
      h3 = (h3 > 255) ? 255 : h3;

      *host_addr = (h0 << 24 | h1 << 16 | h2 << 8 | h3);

      switch (parse_flags)
        {
        case PHF_SOURCE:
          *flags |= ((0x0000ffff & *host_addr) == 0x0002)
            ? CF_INTERNAL_SOURCE : CF_EXTERNAL_SOURCE;
          *flags |= (p < NETWORK_WELL_KNOWN_SERVICES)
            ? CF_SERVER_SOURCE : 0;
          break;

        case PHF_DESTINATION:
          *flags |= ((0x0000ffff & *host_addr) == 0x0002)
            ? CF_INTERNAL_DESTINATION : CF_EXTERNAL_DESTINATION;
          *flags |= (p < NETWORK_WELL_KNOWN_SERVICES)
            ? CF_SERVER_DESTINATION : 0;
          break;

        default:
          break;
        }
      break;

    default:
      perror (PACKAGE ": " __FUNCTION__);
      return 1;
    }

  return 0;
}

/* Initialize the CONNECTION data structure. */
inline void
network_connection_initialize (network_connection_t *connection)
{
  connection->source = 0;
  connection->source_port = 0;
  connection->destination = 0;
  connection->destination_port = 0;
  connection->timestamp = 0.0;
  connection->flags = CF_NONE;
  connection->class = CC_UNDEFINED;

  return;
}

unsigned int
network_next_connection (FILE *stream, network_connection_t *connection)
```

```
{
  char buf[NETWORK_READ_BUF_SIZE];
  char class, remote_host_str[16];
  int local_host = 0, remote_host = 0, local_is_server = 0, service = 0;

  memset (buf, '\0', NETWORK_READ_BUF_SIZE);

  /* Retrieve the next Transmission Control Protocol (TCP) line from the input
     stream.   There should be _no_ damaged connections in the ARTIS files. */
  if (NULL == fgets (buf, NETWORK_READ_BUF_SIZE, stream))
    {
      return 1;
    }

  /* Initialize the connection data. */
  network_connection_initialize (connection);

  /* The general format of this preparsed lines is:
     "classification local_host:remote_host:first_is_server:service"
     The classification may be self (S) or nonself (N). */
  switch (sscanf (buf, "%c %i:%15[0-9.]:%i:%i",
                  &class, &local_host, remote_host_str,
                  &local_is_server, &service))
    {
      unsigned int h0, h1, h2, h3;

    case 5:
      switch (sscanf (remote_host_str, "%u.%u.%u.%u", &h0, &h1, &h2, &h3))
        {
        case 4:
          remote_host = (h0 << 24 | h1 << 16 | h2 << 8 | h3);
          break;
        default:
          perror (PACKAGE ": " __FUNCTION__);
          exit (1);
        }
      break;

    default:
      printf ("%s: %s: buf = '%s'\n", PACKAGE, __FUNCTION__, buf);
      exit (1);
    }

  /* Set the CF_PREPARSED flag to indicate that source is local_host,
     destination is remote_host, destination_port is the service. */
  connection->flags |= CF_PREPARSED;
  connection->source = local_host;
  connection->source_port = local_is_server;
  connection->destination = remote_host;
  connection->destination_port = service;

  /* Determine the classification of the connection.        If CLASS is neither 'S'
     nor 'N', connection->class remains CC_UNDEFINED. */
  switch (class)
    {
    case 'S':
      connection->class = CC_SELF;
```

```
        break;
      case 'N':
        connection->class = CC_NONSELF;
        break;
      }

    return 0;
}

/* Read the next TCP connection from a tcpdump formatted file.   Parse a header
   and store the information in a network_connection_t data structure. */
unsigned int
network_next_tcp_connection (FILE *stream, network_connection_t *connection)
{
  char buf[NETWORK_READ_BUF_SIZE];
  char rest[512], time[32], source[32], destination[32];
  char flag;

  memset (buf, '\0', NETWORK_READ_BUF_SIZE);

  /* Retrieve the next Transmission Control Protocol (TCP) line from the input
     stream. */
  if (NULL == fgets (buf, NETWORK_READ_BUF_SIZE, stream))
    {
      return 1;
    }

  /* If it is a damaged connection header or a fragmented datagram, recurse
     and return the next one. */
  if (strstr (buf, "bytes missing!") || strstr (buf, "frag"))
    {
      return network_next_connection (stream, connection);
    }

  /* Initialize the connection data. */
  network_connection_initialize (connection);

  /* The general format of a TCP (protocol) line is:

         src > dst: flags data-seqno ack window urgent options

     In the TCP line format, src and dst are the source and destination IP
     addresses and ports.   Flags are some combination of S (SYN), F (FIN), P
     (PUSH) or R (RST) or a single '.' (no flags).   The src, dst and flags
     fields are always present. */
  switch (sscanf (buf, "%31s %31[0-9.] > %31[0-9.] : %c %511[^\n]",
                  time, source, destination, &flag, rest))
    {
    case 5:
      /* Convert the time string to a numeral timestamp. */
      connection->timestamp = time_str_to_numeral (time);

      /* Convert the SOURCE IP address and port number string to unsigned ints
         and store the results in CONNECTION. This requires IP version 4, as
         IPv4 addresses are represented by 32 bits. */
      if (network_parse_host_str (source, &connection->source,
                                  &connection->source_port,
                                            &connection->flags,
                                            PHF_SOURCE))
            {
              /* The conversion failed. */
              printf ("%s: %s: source = '%s'\n", PACKAGE, __FUNCTION__,
                      source);
              printf ("%s: %s: buf = '%s'\n", PACKAGE, __FUNCTION__, buf);
              exit (1);
            }

          /* Convert the DESTINATION IP address and port number string to unsigned
             ints and store the results in CONNECTION. */
          if (network_parse_host_str (destination, &connection->destination,
                                            &connection->destination_port,
                                            &connection->flags,
                                            PHF_DESTINATION))
            {
              /* The conversion failed. */
              printf ("%s: %s: destination = '%s'\n", PACKAGE, __FUNCTION__,
                      destination);
              printf ("%s: %s: buf = '%s'\n", PACKAGE, __FUNCTION__, buf);
              exit (1);
            }

          /* Indicate what TCP flag was set in the connection. */
          switch (flag)
            {
            case 'S':
              connection->flags |= CF_TCP_SYN;
              break;
            case 'F':
              connection->flags |= CF_TCP_FIN;
              break;
            case 'P':
              connection->flags |= CF_TCP_PUSH;
              break;
            case 'R':
              connection->flags |= CF_TCP_RST;
              break;
            }

          break;

        default:
          printf ("%s: %s: buf = '%s'\n", PACKAGE, __FUNCTION__, buf);
          exit (1);
        }
  return 0;
}
```

A.9 NETWORK.H

```
/* $Id: network.h,v 1.7 2002/05/09 14:37:50 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */

#ifndef NETWORK_H
#define NETWORK_H

#include <stdio.h>

/* An enumeration of possible conditions related to parsing of TCP packet
    headers. */
typedef enum
{
  CF_NONE = 0x00,
  CF_EXTERNAL_DESTINATION = 0x01 << 0,
  CF_EXTERNAL_SOURCE = 0x01 << 1,
  CF_INTERNAL_DESTINATION = 0x01 << 2,
  CF_INTERNAL_SOURCE = 0x01 << 3,
  CF_SERVER_DESTINATION = 0x01 << 4,
  CF_SERVER_SOURCE = 0x01 << 5,
  CF_TCP_SYN = 0x01 << 6,
  CF_TCP_FIN = 0x01 << 7,
  CF_TCP_PUSH = 0x01 << 8,
  CF_TCP_RST = 0x01 << 9,
  CF_PREPARSED = 0x01 << 10
}
connection_flag_t;

/* An enumeration of the possible classifications of connections. */
typedef enum
{
  CC_UNDEFINED = 0x00,
  CC_SELF = 0x01 << 0,
  CC_NONSELF = 0x01 << 1
}
connection_class_t;

struct _network_connection
{
  unsigned int source;
  unsigned int source_port;
  unsigned int destination;
  unsigned int destination_port;
  double timestamp;
  connection_flag_t flags;
  connection_class_t class;
};
typedef struct _network_connection network_connection_t;

#define NETWORK_WELL_KNOWN_SERVICES 1024

/* Mapping table for network services. */
extern unsigned char network_service_table[];

void network_initialize (void);

unsigned char network_get_service (const unsigned int port);

FILE *network_open (const char *path);
```

```
int network_close (FILE *stream);

unsigned int network_next_connection (FILE *stream,
                                      network_connection_t *connection);

unsigned int network_next_tcp_connection (FILE *stream,
                                          network_connection_t *connection);

#endif /* ! NETWORK_H */
```

A.10 NODE.C

```
/* $Id: node.c,v 1.25 2002/06/04 10:00:53 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif /* HAVE_CONFIG_H */

#include <stdio.h>
#include <stdlib.h>

#include "node.h"

/* A little help structure.   See usage below. */
struct _node_container
{
  ais_node_t *node;
  const bitstring_t *agent;
};
typedef struct _node_container node_container_t;

void
node_initialize (ais_node_t *node, const ais_config_t *config)
{
  unsigned int d;

  node->config = config;

  /* Declare list of immature, mature and memory detectors empty. */
  node->immature_detectors = NULL;
  node->mature_detectors = NULL;
  node->memory_detectors = NULL;

  node->n_mature = 0ULL;
  node->n_memory = 0ULL;

  for (d = 0; d < node->config->n_immature_max; d++)
    {
      node->immature_detectors =
        g_list_prepend (node->immature_detectors,
                        detector_create_random
```

```
                                    (node->config->len, DS_NONE,
                                     config->detector_activation_level_min));

        //bitstring_print (node->detectors[d]->bitstring); printf ("\n");
      }

    return;
}

ais_node_t *
node_new (const ais_config_t *config)
{
    ais_node_t *node = malloc (sizeof (*node));
    if (node == NULL)
      {
        perror (__FUNCTION__ ": virtual memory exhausted");
        exit (1);
      }

    node_initialize (node, config);

    return node;
}

void
node_destroy (ais_node_t *node)
{
    g_list_foreach (node->memory_detectors,
                    (GFunc) detector_destroy_fe_wrapper, NULL);
    g_list_foreach (node->mature_detectors,
                    (GFunc) detector_destroy_fe_wrapper, NULL);
    g_list_foreach (node->immature_detectors,
                    (GFunc) detector_destroy_fe_wrapper, NULL);

    g_list_free (node->memory_detectors);
    g_list_free (node->mature_detectors);
    g_list_free (node->immature_detectors);

    //free (node);

    return;
}

void
print_list_show_element (const GList *list, gpointer *data)
{
    GList *tmp = (GList *) list;
    unsigned int i = 0;

    while (tmp)
      {
        if (data == tmp->data)
          {
            printf ("%d: %p <--\n", i++, tmp->data);
          }
        else
          {
            printf ("%d: %p\n", i++, tmp->data);
          }

        tmp = g_list_next (tmp);
      }
    return;
  }

  void
  print_list (const GList *list)
  {
    GList *tmp = (GList *) list;
    unsigned int i = 0;

    while (tmp)
      {
        printf ("%d: %p\n", i++, tmp->data);

        tmp = g_list_next (tmp);
      }
    return;
  }

  /* Move the detector (referenced by THIS) to the front of LIST. */
  inline GList *
  node_move_detector_to_front (GList *list, GList *this)
  {
    list = g_list_remove_link (list, this);
    return g_list_concat (this, list);
  }

  /* Insert DETECTOR into LIST. If the number of elements in LIST has
     reached its maximum capacity (*N), this is done by applying the
     least reacently used (LRU) principle. */
  inline GList *
  node_insert_detector (GList *list, detector_t *detector,
                        unsigned int *n, const unsigned int n_max)
  {
    detector_t *new = NULL;
    GList *lru = NULL;

    if (*n < n_max)
      {
        /* If DETECTOR is bound to ANTIGEN, NEW will be bound to a
           _copy_ of ANTIGEN. */
        new = detector_clone (detector);
      }
    else
      {
        /* Apply substitution by the least recently used (LRU)
           principle.  As all activated memory detectors are moved to
           the head of the list every time it's activated, the last
           element in the list is the LRU detector.   Start by moving the
           LRU detector to the front of the list. */
        lru = g_list_last (list);
        new = lru->data;
        /* If DETECTOR is bound to ANTIGEN, NEW will be bound to a
           _copy_ of ANTIGEN. */
        detector_recycle_clone (new, detector);
      }

    /* Update NEW's age. */
    new->age = detector->age;
    /* Set NEW's ACTIVATION_AGE. */
    new->activation_age = new->age;

    if (*n < n_max)
      {
        /* Prepend NEW to LIST. */
        list = g_list_prepend (list, new);

        /* Increase the counter of "this type of elements" in LIST.
           Should be equal to g_list_length (LIST). */
        (*n)++;
      }
    else
      {
        list = node_move_detector_to_front (list, lru);
      }

    return list;
  }

  /* The comparison function returns an integer less than, equal to, or
     greater than zero if the first argument is considered to be
     respectively less than, equal to, or greater than the second. */
  inline int
  clone_pool_element_cmp (const void *a, const void *b)
  {
    return ((int) ((clone_pool_element_t *) a)->affinity
            - (int) ((clone_pool_element_t *)b)->affinity);
  }

  /* Reversed version of the above function. */
  inline int
  clone_pool_element_cmp_rev (const void *a, const void *b)
  {
    return (- clone_pool_element_cmp (a, b));
  }

  /* Simulate somatic hypermutation of DETECTOR in the NODE. */
  GList *
  node_hypermutate (GList *list, ais_node_t *node, detector_t *detector)
  {
    register unsigned int i;
    /* If DETECTOR is bound to ANTIGEN during the detector_recycle_clone
       () call, then the clone will be bound to a _copy_ of ANTIGEN.     To
       avoid creating all these copies, we'll temporarioly keep a
       reference to the bound ANTIGEN outside the detector. */
    bitstring_t *antigen = detector->antigen;
    detector->antigen = NULL;

    for (i = 0; i < node->config->clones; i++)
```

```
      {
        /* Clone DETECTOR. */
        detector_recycle_clone (node->clone_pool[i].detector, detector);

        /* Mutate the clone with a probability of P_MUTATE. */
        detector_mutate (node->clone_pool[i].detector, node->config->p_mutate);

        if (node->config->sh_competition)
          {
            node->clone_pool[i].affinity =
              bitstring_contiguous_affinity
              (node->clone_pool[i].detector->bitstring, antigen);
          }
      }

    if (node->config->sh_competition)
      {
        /* Find the AFFINITY level between DETECTOR and ANTIGEN. */
        unsigned int affinity =
          bitstring_contiguous_affinity (detector->bitstring,
                                         antigen);

        /* Sort the cloned detectors, according to the strength of their
           affinity to the ANTIGEN.  The sorted list will be arranged
           from high to low affinity values. */
        qsort (node->clone_pool, node->config->clones,
               sizeof (*(node->clone_pool)), clone_pool_element_cmp_rev);

        /* If SH_COMPETITION is enabled, select the clones with the
           highest affinity (minimum equal to AFFINITY) for AGENT.  No
           more than CLONES_SELECT will be selected. */
        i = 0;
        while (i < node->config->clones_select)
          {
            if (node->clone_pool[i].affinity >= affinity)
              {
                list = node_insert_detector (list,
                                             node->clone_pool[i].detector,
                                             &node->n_memory,
                                             node->config->n_memory_max);
              }
            else
              {
                /* As the CLONE_POOL is an ordered list (high to low),
                   if the current clone's affinity is below the
                   threshold AFFINITY, none of its successors will be
                   either. */
                i = node->config->clones_select;
              }
            i++;
          }
      }
    else                            /* ! SH_COMPETITION. */
      {
        /* If SH_COMPETITION is not enabled, simply randomly select the
           CLONES_SELECT clones from CLONE_POOL and insert them into
           LIST. */
      for (i = 0; i < node->config->clones_select; i++)
        {
          /* Note that none of the clones are bound to the antigen
             inside this function.    Thus, there's no need to unbind
             before insertion into LIST. */
          list = node_insert_detector (list,
                                       node->clone_pool[i].detector,
                                       &node->n_memory,
                                       node->config->n_memory_max);
        }
    }

  /* Remember to set the reference to ANTIGEN back again. */
  detector->antigen = antigen;

  return list;
}

/* Update the statistics concerning true and false positives and
   negatives. */
inline void
node_update_stats (ais_node_info_t *info, const unsigned char matched,
                   const connection_class_t class)
{
  switch (class)
    {
    case CC_SELF:
      matched ? info->false_positives++ : info->true_negatives++;
      break;

    case CC_NONSELF:
      matched ? info->true_positives++ : info->false_negatives++;
      break;

    default:
      break;
    }

  return;
}

inline void
node_expose_immature_to_agent (detector_t *detector, node_container_t *data)
{
  if (bitstring_r_contiguous (detector->bitstring, data->agent,
                              data->node->config->r))
    {
      /* DETECTOR matched AGENT. */
      node_update_stats (&data->node->info, TRUE,
                         data->node->config->current_class);

      /* Increase the activation level of DETECTOR. */
      detector_activation_increase
        (detector, data->node->config->detector_activation_inc,
         data->node->config->detector_activation_level_max);

      /* FIXME: uncomment? */
```

```
        detector->state |= DS_ACTIVE;

        if ((detector->state & (DS_ACTIVE | DS_MATURE)) == DS_ACTIVE)
          {
            /* Active and immature; i.e., it recognized self.      Thus, the
               detector is killed. */
            detector->state |= DS_DEAD;

            data->node->info.premature_deaths++;
          }
        else
          {
            /* If DETECTOR is mature and active but haven't been
               classified yet, then set its costimulation signal. */
            if (((detector->state
                  & (DS_ACTIVE | DS_MATURE | DS_COSTIMULATION_SELF
                     | DS_COSTIMULATION_NONSELF)))
                == (DS_ACTIVE | DS_MATURE))
              {
                detector->state |=
                  (data->node->config->current_class == CC_NONSELF)
                  ? DS_COSTIMULATION_NONSELF : DS_COSTIMULATION_SELF;
                detector->activation_age = detector->age;
                data->node->info.activations++;

                if (data->node->config->sh_competition
                    && ((detector->state & DS_COSTIMULATION_NONSELF)
                        == DS_COSTIMULATION_NONSELF))
                  {
                    /* Bind to (a copy of) the AGENT. */
                    detector_antigen_bind (detector, data->agent);
                  }
              }
          }
      }
    else
      {
        /* DETECTOR didn't match AGENT. */
        node_update_stats (&data->node->info, FALSE,
                           data->node->config->current_class);

        detector_activation_decrease
          (detector, data->node->config->detector_activation_dec,
           data->node->config->detector_activation_level_min);
      }

    /* Check if we've finally become DS_MATURE. */
    if ((detector->state & (DS_ACTIVE | DS_MATURE)) == DS_MATURE)
      {
        data->node->mature_detectors =
          node_insert_detector (data->node->mature_detectors, detector,
                                &data->node->n_mature,
                                data->node->config->n_mature_max);

        detector->state |= DS_DEAD;
      }
```

```
  /* Age the DETECTOR. */
  detector_age (detector, data->node->config->detector_immature_period,
                data->node->config->detector_lifetime);

  if ((detector->state & DS_DEAD) == DS_DEAD)
    {
      /* Immature detector is dead.  Recycle it (give it a random
         bitstring value and set its state to DS_NONE). */
      if (detector->antigen != NULL)
        {
          detector_antigen_unbind (detector);
        }

      detector_recycle_random
        (detector, data->node->config->len, DS_NONE,
         data->node->config->detector_activation_level_min);
    }

  return;
}

inline void
node_expose_mature_to_agent (detector_t *detector, node_container_t *data)
{
  if (bitstring_r_contiguous (detector->bitstring, data->agent,
                              data->node->config->r))
    {
      /* DETECTOR matched AGENT. */
      node_update_stats (&data->node->info, TRUE,
                         data->node->config->current_class);

      data->node->info.mature_matched = TRUE;

      /* Increase the activation level of DETECTOR.  Note that we're
         using DETECTOR_ACTIVATION_INC as the increase rate. */
      detector_activation_increase
        (detector, data->node->config->detector_activation_inc,
         data->node->config->detector_activation_level_max);

      /* If DETECTOR is mature and active but haven't been classified
         yet, then set its costimulation signal. */
      if (((detector->state
            & (DS_ACTIVE | DS_MATURE | DS_COSTIMULATION_SELF
               | DS_COSTIMULATION_NONSELF)))
          == (DS_ACTIVE | DS_MATURE))
        {
          detector->state |=
            (data->node->config->current_class == CC_NONSELF)
            ? DS_COSTIMULATION_NONSELF : DS_COSTIMULATION_SELF;
          detector->activation_age = detector->age;
          data->node->info.activations++;

          if (data->node->config->sh_competition
              && ((detector->state & DS_COSTIMULATION_NONSELF)
                  == DS_COSTIMULATION_NONSELF))
            {
              /* Bind to (a copy of) the AGENT. */
              detector_antigen_bind (detector, data->agent);
            }
        }
    }
  else
    {
      /* DETECTOR didn't match AGENT. */
      node_update_stats (&data->node->info, FALSE,
                         data->node->config->current_class);

      /* Decrease activation level.  Note that we're using
         DETECTOR_ACTIVATION_DEC as the decrease rate. */
      detector_activation_decrease
        (detector, data->node->config->detector_activation_dec,
         data->node->config->detector_activation_level_min);
    }

  return;
}

inline void
node_expose_memory_to_agent (detector_t *detector, node_container_t *data)
{
  if (bitstring_r_contiguous (detector->bitstring, data->agent,
                              data->node->config->r))
    {
      /* DETECTOR matched AGENT. */
      node_update_stats (&data->node->info, TRUE,
                         data->node->config->current_class);

      data->node->info.memory_matched = TRUE;

      /* Increase the activation level of DETECTOR.  Note that we're
         using MEMORY_DETECTOR_ACTIVATION_INC as the increase rate. */
      detector_activation_increase
        (detector, data->node->config->memory_detector_activation_inc,
         data->node->config->detector_activation_level_max);

      /* If DETECTOR is mature and active but haven't been classified
         yet, then set its costimulation signal. */
      if (((detector->state
            & (DS_ACTIVE | DS_MATURE | DS_COSTIMULATION_SELF
               | DS_COSTIMULATION_NONSELF)))
          == (DS_ACTIVE | DS_MATURE))
        {
          detector->state |=
            (data->node->config->current_class == CC_NONSELF)
            ? DS_COSTIMULATION_NONSELF : DS_COSTIMULATION_SELF;
          detector->activation_age = detector->age;
          data->node->info.activations++;

          if (data->node->config->sh_competition
              && ((detector->state & DS_COSTIMULATION_NONSELF)
                  == DS_COSTIMULATION_NONSELF))
            {
              /* Bind to (a copy of) the AGENT. */
              detector_antigen_bind (detector, data->agent);
              }
          }
      }
    else
      {
        /* DETECTOR didn't match AGENT. */
        node_update_stats (&data->node->info, FALSE,
                           data->node->config->current_class);

        /* Decrease activation level.     Note that we're using
           MEMORY_DETECTOR_ACTIVATION_DEC as the decrease rate. */
        detector_activation_decrease
          (detector, data->node->config->memory_detector_activation_dec,
           data->node->config->detector_activation_level_min);
      }

    return;
  }

  inline void
  node_detector_age_fe (detector_t *detector, const ais_node_t *node)
  {
    detector_age (detector, node->config->detector_immature_period,
                  node->config->detector_lifetime);

    return;
  }

  inline GList *
  node_process_detectors (GList *list, unsigned int *n, ais_node_t *node,
                          unsigned char proc_memory)
  {
    GList *tmp = list;
    unsigned int m = 0;

    //unsigned char *str = proc_memory ? "mem: " : "mat: ";

    while (tmp != NULL)
      {
        detector_t *detector = tmp->data;
        GList *next = g_list_next (tmp);

        /* First, see if DETECTOR has done any detections at all. */
        if ((detector->state & (DS_ACTIVE | DS_MATURE))
            == (DS_ACTIVE | DS_MATURE))
          {
            /* See if we're done waiting for costimulation. */
            if ((detector->age - detector->activation_age)
                >= node->config->costimulation_delay)
              {
                if ((detector->state & DS_COSTIMULATION_SELF)
                    == DS_COSTIMULATION_SELF)
                  {
                    /* If the activating agent was self, no
                       costimualtion (signal 2) is received, and
                       DETECTOR dies. */
                    detector->state |= DS_DEAD;
            /* Increase the number of detectors which received
               signal 1 only. */
            node->info.sig_1_only++;
          }
        else
          {
            /* If the activating agent was nonself, signal 2 is
               received, DETECTOR becomes a memory detector and
               possibly undergoes somatic hypermutation. */
            if ((detector->state & DS_COSTIMULATION_NONSELF)
               == DS_COSTIMULATION_NONSELF)
              {
                /* Clear the following status flags. */
                detector->state &= ~(DS_ACTIVE | DS_COSTIMULATION_SELF
                                     | DS_COSTIMULATION_NONSELF);

                /* Increase the number of detectors which
                   received both signal 1 and signal 2
                   (costimulation). */
                node->info.sig_1_and_2++;

                /* Set the state of the detector to
                   DS_MEMORY. */
                detector->state |= DS_MEMORY;

                /* See if we're processing memory detectors. */
                if (proc_memory == TRUE)
                  {
                    /* Move the element to the front of the
                       list.   This is done to support the LRU
                       substitution. */
                    list = node_move_detector_to_front (list, tmp);
                  }
                else
                  {
                    /* Insert the detector into the memory
                       detectors. */
                    node->memory_detectors =
                      node_insert_detector (node->memory_detectors,
                                            detector,
                                            &node->n_memory,
                                            node->config->n_memory_max);
                  }

                /* If the SOMATIC_HYPERMUTATION flag is set,
                   then simulate SH. */
                if (node->config->somatic_hypermutation)
                  {
                    /* Simulate somatic hypermutation (SH). */
                    if (proc_memory == TRUE)
                      {
                        list = node_hypermutate (list,
                                                 node, detector);
                      }
                    else
                      {
```

```
                                    node->memory_detectors =
                                      node_hypermutate (node->memory_detectors,
                                                        node, detector);

                                    /* As the DETECTOR is cloned in the
                                       above statement, _this_ one may be
                                       removed from this list by setting its
                                       state to DS_DEAD. */
                                    detector->state |= DS_DEAD;
                                  }
                              }
                          }
                      }
                  }
              }

            m++;

            /* Age DETECTOR. */
            detector_age (detector, node->config->detector_immature_period,
                          node->config->detector_lifetime);

            /* Iterate further in the list, before possibly removing
               DETECTOR. */
            tmp = next;

            if ((detector->state & DS_DEAD) == (DS_DEAD))
              {
                /* Remove the DETECTOR from the list of mature detectors. */
                list = g_list_remove (list, detector);

                /* We should definitely unbind the ANTIGEN from the DETECTOR
                   it has been bound to (if any) before DETECTOR is
                   destroyed. */
                if (detector->antigen != NULL)
                  {
                    detector_antigen_unbind (detector);
                  }

                detector_destroy (detector);

                /* Decrease the mature/detector cell counter. */
                --(*n);
              }

            /* A safety fuse to avoid any possible infinite loop. */
            if (m > 1000)
              {
                fprintf (stderr, PACKAGE " " VERSION ":" __FUNCTION__
                         ": exited an infinite loop.");
                exit (1);
              }
          }

        return list;
      }
```

```
/* Simulate the insertion of AGENT into the IS. */
inline void
node_insert_agent (ais_node_t *node, const bitstring_t *agent)
{
  node_container_t data;

  data.agent = agent;
  data.node = node;

  /* See if any memory detectors match AGENT. */
  node->info.memory_matched = FALSE;

  g_list_foreach (node->memory_detectors,
                 (GFunc) node_expose_memory_to_agent, &data);

  /* See if any mature detectors match AGENT. */
  node->info.mature_matched = FALSE;

  g_list_foreach (node->mature_detectors,
                 (GFunc) node_expose_mature_to_agent, &data);

  /* If neither memory nor mature detectors matched AGENT, it is probably
     self.   This protective behavior tries to simulate the thymus. */
  if (node->config->thymus)
    {
      if ((node->info.memory_matched == FALSE)
          && (node->info.mature_matched == FALSE))
        {
          g_list_foreach (node->immature_detectors,
                         (GFunc) node_expose_immature_to_agent, &data);
        }
    }
  else
    {
      g_list_foreach (node->immature_detectors,
                     (GFunc) node_expose_immature_to_agent, &data);
    }

  node->mature_detectors =
    node_process_detectors (node->mature_detectors, &(node->n_mature), node,
                            FALSE);

  node->memory_detectors =
    node_process_detectors (node->memory_detectors, &(node->n_memory), node,
                            TRUE);

  return;
}
```

A.11 NODE.H

```
/* $Id: node.h,v 1.9 2002/05/27 18:44:45 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */
```

```
#ifndef NODE_H
#define NODE_H

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif /* HAVE_CONFIG_H */

#include <glib.h>

#include "ais_config.h"
#include "detector.h"

struct _ais_node_info
{
  unsigned int premature_deaths;
  unsigned int true_positives;   /* Matching nonself. */
  unsigned int true_negatives;   /* Not matching self. */
  unsigned int false_positives;  /* Matching self. */
  unsigned int false_negatives;  /* Not matching nonself. */
  unsigned int activations;
  unsigned int sig_1_and_2;
  unsigned int sig_1_only;

  unsigned char memory_matched; /* TRUE if any memory detector has matched an
                                   agent, FALSE otherwise. */
  unsigned char mature_matched; /* TRUE if any mature detector has matched an
                                   agent, FALSE otherwise. */
};
typedef struct _ais_node_info ais_node_info_t;

struct _clone_pool_element
{
  detector_t *detector;
  unsigned int affinity;
};
typedef struct _clone_pool_element clone_pool_element_t;

struct _ais_node
{
  /* Doubly-linked lists. */
  GList *immature_detectors;
  GList *mature_detectors;
  GList *memory_detectors;

  unsigned int n_mature;         /* The number of mature detectors currently in
                                    MATURE_DETECTORS. */
  unsigned int n_memory;         /* The number of memory detectors currently in
                                    MEMORY_DETECTORS. */
  ais_node_info_t info;
  const ais_config_t *config;
  clone_pool_element_t *clone_pool;
  //detector_t **clone_pool;
};
typedef struct _ais_node ais_node_t;

void node_initialize (ais_node_t *node, const ais_config_t *config);
```

```
ais_node_t *node_new (const ais_config_t *config);

void node_destroy (ais_node_t *node);

void node_insert_agent (ais_node_t *node, const bitstring_t *agent);

void node_response (ais_node_t *node);

#endif /* ! NODE_H */
```

A.12 MAIN.C

```
/* $Id: main.c,v 1.20 2002/05/27 18:44:45 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif /* HAVE_CONFIG_H */

#include "ais.h"
#include "network.h"

#ifdef MTRACE
#include <mcheck.h>
#endif /* MTRACE */

#include <stdlib.h>
#include <string.h>
#include <getopt.h>

void
main_print_usage (FILE *stream, const char *program_name)
{
  fprintf (stream, "Usage:  %s [ options ] [ inputfile . . . ]\n", program_name);
  fprintf (stream,
           "  -h, --help             "
           "Display this usage information.\n"
           "  -i, --info             "
           "Only show the current configuration and exit.\n"
           "  -t, --test             "
           "Perform a match test.\n"
           "  -r R, --r-value=R      "
           "Set the r of the r-contiguous-bits algorithm.\n"
           "  -n N, --nodes=N        "
           "Set the number of nodes.\n"
           "  -I N, --immature-detectors=N\n                   "
           "Set the number of immature detectors per node.\n"
           "  -N N, --mature-detectors=M "
           "Set the maximum number of mature (naive) agents.\n"
           "  -M N, --memory-detectors=M "
           "Set the maximum number of mature agents.\n"
           "  -l N, --detector-lifetime=N \n                   "
```

```
          "Set the length of a detectors lifetime.\n"
          "  -c N, --detector-childhood=N\n                     "
          "Set the duration of a detectors childhood.\n"
          "  -a R, --activation-inc=R "
          "Set the factor for increasing the detector\n"
          "                           "
          "activation.\n"
          "  -d R, --activation-dec=R "
          "Set the factor for decreasing the detector\n"
          "                           "
          "activation.\n"
          "  -D N, --costimulation-delay \n                    "
          "Set the length of the period from activation\n"
          "                           "
          "to costimulation is received.\n"
          "  -f F, --format=F         "
          "Use TCP or AIS as file format. Default is AIS.\n"
          "  -p S, --node-ids=S       "
          "Read node IDs from file S. (Switched LAN.)\n"
          "  -y, --somatic-hypermutation \n                   "
          "Perform somatic hypermutation (SH).\n"
          "  -m, --sh-p-mutate=P      "
          "Let P be the probability of mutation under SH.\n"
          "  -u, --sh-clones=N        "
          "Generate N clones under SH.\n"
          "  -v, --sh-clones-select=N "
          "Let N of the clones survive after SH.\n"
          "  -C, --sh-competition     "
          "Simulate affinity competition under SH.\n"
          "  -z, --thymus             "
          "Perform thymus simulation.\n"
          );

  return;
}

void
main_print_info (FILE *stream, const ais_config_t *config)
{
  unsigned int f;

  fprintf (stream,
          "# " PACKAGE ", version " VERSION "\n"
          "#\n# Current configuration:\n"
          "# somatic_hypermutation = %d\n"
          "# sh_competition = %d\n"
          "# thymus = %d\n"
          "# p_mutate = %g\n"
          "# clones = %d\n"
          "# clones_select = %d\n"
          "# nodes = %i\n"
          "# n_immature_max = %i\n"
          "# n_mature_max = %i\n"
          "# n_memory_max = %i\n"
          "# len = %i\n"
          "# r = %i\n"
          "# detector_lifetime = %i\n"
```

```
           "# detector_immature_period = %i\n"
           "# detector_activation_level_max = %g\n"
           "# detector_activation_level_min = %g\n"
           "# detector_activation_inc = %g\n"
           "# detector_activation_dec = %g\n"
           "# costimulation_delay = %i\n"
           "# match_test = %d\n"
           "# file_format = %s\n",
           config->somatic_hypermutation,
           config->sh_competition,
           config->thymus,
           config->p_mutate, config->clones, config->clones_select,
           config->nodes, config->n_immature_max,
           config->n_mature_max, config->n_memory_max,
           config->len, config->r, config->detector_lifetime,
           config->detector_immature_period,
           config->detector_activation_level_max,
           config->detector_activation_level_min,
           config->detector_activation_inc, config->detector_activation_dec,
           config->costimulation_delay,
           config->match_test,
           ((config->file_format == AIS_FILE_FORMAT_TCP) ? "TCP" : "AIS"));

  if (config->node_ids != NULL)
    {
      fprintf (stream, "# switched_lan = 1\n");
      fprintf (stream, "# node_ids = '%s'\n", config->node_ids);
    }
  else
    {
      fprintf (stream, "# switched_lan = 0\n");
    }

  for (f = 0; f < config->file_names; f++)
    {
      fprintf (stream, "# file %d = '%s'\n", f, config->file_name_list[f]);
    }

  return;
}

int
main (int argc, char *argv[])
{
  /* A string listing of valid short option letters. */
  const char *short_options = "htr:n:I:N:M:l:c:f:p:ym:u:v:Cz";

  /* An array describing valid long options and corresponding short
     options. */
  const struct option long_options[] =
    {
      {"help", no_argument, NULL, 'h'},
      {"info", no_argument, NULL, 'i'},
      {"test", no_argument, NULL, 't'},
      {"r-value", required_argument, NULL, 'r'},
      {"nodes", required_argument, NULL, 'n'},
      {"immature-detectors", required_argument, NULL, 'I'},
      {"mature-detectors", required_argument, NULL, 'N'},
      {"memory-detectors", required_argument, NULL, 'M'},
      {"detector-lifetime", required_argument, NULL, 'l'},
      {"detector-childhood", required_argument, NULL, 'c'},
      {"activation-inc", required_argument, NULL, 'a'},
      {"activation-dec", required_argument, NULL, 'd'},
      {"costimulation-delay", required_argument, NULL, 'D'},
      {"format", required_argument, NULL, 'f'},
      {"node-ids", required_argument, NULL, 'p'},
      {"somatic-hypermutation", no_argument, NULL, 'y'},
      {"sh-p-mutate", required_argument, NULL, 'm'},
      {"sh-clones", required_argument, NULL, 'u'},
      {"sh-clones-select", required_argument, NULL, 'v'},
      {"sh-competition", no_argument, NULL, 'C'},
      {"thymus", no_argument, NULL, 'z'},
      {NULL, 0, NULL, 0}
    };
  unsigned int f;
  unsigned int show_info = FALSE;
  ais_t *ais = NULL;
  int next_option;

#ifdef MTRACE
  mtrace ();                    /* Turn on malloc tracing. */
#endif /* MTRACE */

  /* Create a new artificial immune system (AIS).    The fields of the ais
     struct will all be set to default values. */
  ais = ais_new ();

  /* Parse the command line options. */
  do
    {
      next_option = getopt_long (argc, argv, short_options,
                                 long_options, NULL);

      switch (next_option)
        {
        case 'h':               /* -h or –help. */
          main_print_usage (stderr, argv[0]);
          exit (1);
          break;

        case 't':               /* -t or –test. */
          /* Perform a benchmark of the r_contiguos_match algorithm. */
          ais->config->match_test = TRUE;
          break;

        case 'r':               /* -r or –r-value. */
          /* Set the r of the r-contiguous-bits algorithm. */
          ais->config->r = (unsigned int) atoi (optarg);
          break;

        case 'n':               /* -n or –nodes. */
          /* Set the number of nodes. */
          ais->config->nodes = (unsigned int) atoi (optarg);
          break;
    case 'I':                 /* -I or --immature-detectors. */
      /* Set the number of _immature_ detectors per node. */
      ais->config->n_immature_max = (unsigned int) atoi (optarg);
      break;

    case 'N':                 /* -N or --mature-detectors. */
      ais->config->n_mature_max = (unsigned int) atoi (optarg);
      break;

    case 'M':                 /* -M or --memory-detectors. */
      /* Set the number of _memory_ detectors per node. */
      ais->config->n_memory_max = (unsigned int) atoi (optarg);
      break;

    case 'l':                 /* -l or --detector-lifetime. */
      /* Set the length of a detectors lifetime. */
      ais->config->detector_lifetime = (unsigned int) atoi (optarg);
      break;

    case 'c':                 /* -c or --detector-childhood. */
      /* Set the duration of a detectors childhood. */
      ais->config->detector_immature_period = (unsigned int) atoi (optarg);
      break;

    case 'a':                 /* -a or --activation-inc. */
      ais->config->detector_activation_inc = atof (optarg);
      break;

    case 'd':                 /* -d or --activation-dec. */
      ais->config->detector_activation_dec = atof (optarg);
      break;

    case 'D':                 /* -D or --costimulation-delay. */
      ais->config->costimulation_delay = atoi (optarg);
      break;

    case 'i':                 /* -i or --info. */
      /* Only show the current configuration and exit. */
      show_info = TRUE;
      break;

    case 'f':                 /* -f or --format. */
      if (0 == strcmp ("TCP", optarg))
        {
          ais->config->file_format = AIS_FILE_FORMAT_TCP;
        }
      else
        {
          ais->config->file_format = AIS_FILE_FORMAT_AIS;
        }
      break;

    case 'p':                 /* -p or --node-ids. */
      ais->config->node_ids = optarg;
      break;

```

```
        case 'y':               /* -y or --somatic-hypermutation. */
          ais->config->somatic_hypermutation = 1;
          break;

        case 'm':               /* -m or --p-mutate. */
          ais->config->p_mutate = atof (optarg);
          break;

        case 'u':               /* -u or --clones. */
          ais->config->clones = atoi (optarg);
          break;

        case 'v':               /* -v or --clones-select. */
          ais->config->clones_select = atoi (optarg);
          break;

        case 'C':               /* -C or --sh-competition. */
          ais->config->sh_competition = 1;
          break;

        case 'z':               /* -z or --thymus. */
          ais->config->thymus = 1;
          break;
        }
    }
  while (next_option != -1);

  /* Retrieve the names of the files to be used as input.      The OPTIND points
     to the first non-option argument. */
  for (f = optind; f < argc; f++)
    {
      ais->config->file_names++;
    }

  if (ais->config->file_names == 0)
    {
      show_info = TRUE;
    }
  else
    {
      ais->config->file_name_list =
        (unsigned char **) malloc (sizeof (*(ais->config->file_name_list))
                                  * ais->config->file_names);

      for (f = optind; f < argc; f++)
        {
          ais->config->file_name_list[f - optind] = argv[f];
        }
    }

  if (ais->config->node_ids != NULL)
    {
      ais_generate_node_map (ais);
    }

  ais_initialize (ais);
  main_print_info (stdout, ais->config);

  if (ais->config->match_test)
    {
      ais_match_test (ais);
    }

  if (show_info)
    {
      exit (0);
    }

  if (ais->config->node_ids != NULL)
    {
      ais_run_switched (ais);
    }
  else
    {
      ais_run_broadcast (ais);
    }

  ais_destroy (ais);

  return 0;
}
```

A.13 FAST_LOG_PARSE.C

```
/* $Id: fast_log_parse.c,v 1.21 2002/06/12 07:05:33 mtr Exp $
 *
 * Copyright (C) 2002 by Martin Thorsen Ranang <mtr@ranang.org>
 */

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif /* HAVE_CONFIG_H */

#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "ais_config.h"
#include "fast_log_parse.h"

#define FGETS_BUF_SIZE 4096

#ifndef max
#define max(X, Y) \
     ({ typeof (X) x_ = (X); \
     typeof (Y) y_ = (Y); \
     (x_ > y_) ? x_ : y_; })
#endif /* ! max. */
```

```
#ifndef min
#define min(X, Y) \
     ({ typeof (X) x_ = (X); \
     typeof (Y) y_ = (Y); \
     (x_ < y_) ? x_ : y_; })
#endif /* ! min. */

#ifdef _LIBC
#define MEMPCPY(d, s, n) __mempcpy (d, s, n)
#else
#ifndef HAVE_MEMPCPY
#define MEMPCPY(d, s, n) ((void *) ((char *) memcpy (d, s, n) + (n)))
#endif
#endif

FILE *
flp_file_open (char *file_name)
{
  FILE *stream = NULL;

  stream = fopen (file_name, "r");
  if (stream == NULL)
    {
      perror (PACKAGE ": " __FILE__ ": " __FUNCTION__ ": open ()");
      exit (1);
    }

  return stream;
}

void
flp_file_close (FILE *stream)
{
  fclose (stream);

  return;
}

unsigned int
flp_file_is_self (const ais_config_t *config, const unsigned int file)
{
  if (config->file_name_list[file][0] == 's')
    {
      return TRUE;
    }
  else
    {
      return FALSE;
    }
}

void
flp_config_print (FILE *stream, const ais_config_t *config)
{
  unsigned int f;

  fprintf (stream,
              "# " PACKAGE ", version " VERSION "\n"
              "#\n# Current configuration:\n"
              "# somatic_hypermutation = %d\n"
              "# sh_competition = %d\n"
              "# thymus = %d\n"
              "# p_mutate = %g\n"
              "# clones = %d\n"
              "# clones_select = %d\n"
              "# nodes = %i\n"
              "# n_immature_max = %i\n"
              "# n_mature_max = %i\n"
              "# n_memory_max = %i\n"
              "# len = %i\n"
              "# r = %i\n"
              "# detector_lifetime = %i\n"
              "# detector_immature_period = %i\n"
              "# detector_activation_level_max = %g\n"
              "# detector_activation_level_min = %g\n"
              "# detector_activation_inc = %g\n"
              "# detector_activation_dec = %g\n"
              "# costimulation_delay = %i\n"
              "# match_test = %d\n"
              "# file_format = %s\n",
              config->somatic_hypermutation,
              config->sh_competition,
              config->thymus,
              config->p_mutate, config->clones, config->clones_select,
              config->nodes, config->n_immature_max,
              config->n_mature_max, config->n_memory_max,
              config->len, config->r, config->detector_lifetime,
              config->detector_immature_period,
              config->detector_activation_level_max,
              config->detector_activation_level_min,
              config->detector_activation_inc, config->detector_activation_dec,
              config->costimulation_delay,
              config->match_test,
              ((config->file_format == AIS_FILE_FORMAT_TCP) ? "TCP" : "AIS"));

  if (config->node_ids != NULL)
    {
      fprintf (stream, "# switched_lan = 1\n");
      fprintf (stream, "# node_ids = '%s'\n", config->node_ids);
    }
  else
    {
      fprintf (stream, "# switched_lan = 0\n");
    }

  for (f = 0; f < config->file_names; f++)
    {
      fprintf (stream, "# file %d = '%s' [%s]\n", f,
               config->file_name_list[f],
               flp_file_is_self (config, f) ? "self" : "nonself");
    }

  return;
}
```

```
unsigned int
flp_config_read (ais_config_t *config, FILE *stream)
{
  unsigned int done = FALSE;
  config->file_name_list = NULL;

  while (! done)
    {
      char buf[FGETS_BUF_SIZE];
      char *ptr, *input;

      if (NULL == fgets (buf, FGETS_BUF_SIZE, stream))
        {
          return 1;
        }

      input = buf;

      ptr = strsep (&input, "=");
      if (input != NULL)
        {
          if (strcmp ("# somatic_hypermutation ", ptr) == 0)
            {
              config->somatic_hypermutation = atoi (input);
            }
          else if (strcmp ("# sh_competition ", ptr) == 0)
            {
              config->sh_competition = atoi (input);
            }
          else if (strcmp ("# thymus ", ptr) == 0)
            {
              config->thymus = atoi (input);
            }
          else if (strcmp ("# p_mutate ", ptr) == 0)
            {
              config->p_mutate = atof (input);
            }
          else if (strcmp ("# clones ", ptr) == 0)
            {
              config->clones = atoi (input);
            }
          else if (strcmp ("# clones_select ", ptr) == 0)
            {
              config->clones_select = atoi (input);
            }
          else if (strcmp ("# nodes ", ptr) == 0)
            {
              config->nodes = atoi (input);
            }
          else if (strcmp ("# n_immature_max ", ptr) == 0)
            {
              config->n_immature_max = atoi (input);
            }
          else if (strcmp ("# n_mature_max ", ptr) == 0)
            {
              config->n_mature_max = atoi (input);
      }
    else if (strcmp ("# n_memory_max ", ptr) == 0)
      {
        config->n_memory_max = atoi (input);
      }
    else if (strcmp ("# len ", ptr) == 0)
      {
        config->len = atoi (input);
      }
    else if (strcmp ("# r ", ptr) == 0)
      {
        config->r = atoi (input);
      }
    else if (strcmp ("# detector_lifetime ", ptr) == 0)
      {
        config->detector_lifetime = atoi (input);
      }
    else if (strcmp ("# detector_immature_period ", ptr) == 0)
      {
        config->detector_immature_period = atoi (input);
      }
    else if (strcmp ("# detector_immature_period ", ptr) == 0)
      {
        config->detector_immature_period = atoi (input);
      }
    else if (strcmp ("# detector_activation_level_max ", ptr) == 0)
      {
        config->detector_activation_level_max = atof (input);
      }
    else if (strcmp ("# detector_activation_level_min ", ptr) == 0)
      {
        config->detector_activation_level_min = atof (input);
      }
    else if (strcmp ("# detector_activation_inc ", ptr) == 0)
      {
        config->detector_activation_inc = atof (input);
      }
    else if (strcmp ("# detector_activation_dec ", ptr) == 0)
      {
        config->detector_activation_dec = atof (input);
      }
    else if (strcmp ("# costimulation_delay ", ptr) == 0)
      {
        config->costimulation_delay = atoi (input);
      }
    /* We don't care if it was a match_test or what file_format the
       simulation used. */
    else if (strcmp ("# switched_lan ", ptr) == 0)
      {
        if (atoi (input) == 0)
          {
            config->node_ids = NULL;
          }
      }
    /* Read the file names.   Their 'self' and 'nonself' prefixes are
       used to determine the correctnes of any ACTIVATIONS occuring in
       the log. */
```

```
          else if (strncmp ("# file ", ptr, 7) == 0)
            {
              char *tmp = strsep (&input, "'");
              tmp = strsep (&input, "'");

              config->file_name_list =
                (unsigned char **) realloc (config->file_name_list,
                                            sizeof (*config->file_name_list)
                                            * (++config->file_names));
              config->file_name_list[config->file_names - 1] = strdup (tmp);
            }
          /* When the "# time " string is reached, we're done. */
          else if (strcmp ("# time ", ptr) == 0)
            {
              done = TRUE;
            }
        }
    }
  return 0;
}

unsigned long long
flp_analysis_min (const unsigned long long *data, const unsigned int len)
{
  unsigned int n;
  unsigned long long res = data[0];

  for (n = 0; n < len; n++)
    {
      res = min (res, data[n]);
    }

  return res;
}

unsigned long long
flp_analysis_max (const unsigned long long *data, const unsigned int len)
{
  unsigned int n;
  unsigned long long res = data[0];

  for (n = 0; n < len; n++)
    {
      res = max (res, data[n]);
    }

  return res;
}

long double
flp_analysis_avg (const unsigned long long *data, const unsigned int len)
{
  unsigned int n;
  unsigned long long sum = 0ULL;

  for (n = 0; n < len; n++)
    {
      sum += data[n];
    }

  return (sum / (long double) len);
}

void
flp_analysis_print_avg_min_max (const ais_config_t *config,
                                const unsigned long long *val)
{
  printf ("%.5Lf %qu %qu",
          flp_analysis_avg (val, config->nodes),
          flp_analysis_min (val, config->nodes),
          flp_analysis_max (val, config->nodes));
  return;

}

void
flp_analysis_responses_print (const statistics_t *stats,
                              const ais_config_t *config)
{
  unsigned int f;

  for (f = 0; f < config->file_names; f++)
    {
      printf ("(%d, %d) ", stats->false_alarms[f], stats->true_alarms[f]);
    }

  return;
}

void
flp_analysis_info_print (const ais_config_t *config)
{
  printf ("%d %d %d %d %d %d %d %d %d %d %g %g %d %d %d:",
          config->nodes, config->clones_select, config->n_immature_max,
          config->n_mature_max,
          config->n_memory_max, config->len, config->r,
          config->detector_immature_period, config->detector_lifetime,
          config->costimulation_delay,
          config->detector_activation_inc, config->detector_activation_dec,
          config->somatic_hypermutation, config->sh_competition,
          config->thymus);

  return;
}

void
flp_analysis_rough_print (const statistics_t *stats, const ais_config_t *config)
{
  flp_analysis_print_avg_min_max (config, stats->premature_deaths);
  printf (", ");
  flp_analysis_print_avg_min_max (config, stats->false_negatives);
  printf (" ");
  printf ("%qu, ", stats->b_false_negatives);
  flp_analysis_print_avg_min_max (config, stats->true_negatives);
  printf (" ");
  printf ("%qu, ", stats->b_true_negatives);
  flp_analysis_print_avg_min_max (config, stats->false_positives);
  printf (" ");
  printf ("%qu, ", stats->b_false_positives);
  flp_analysis_print_avg_min_max (config, stats->true_positives);
  printf (" ");
  printf ("%qu, ", stats->b_true_positives);
  flp_analysis_print_avg_min_max (config, stats->activations);
  printf (", ");
  flp_analysis_print_avg_min_max (config, stats->sig_1_only);
  printf (", ");
  flp_analysis_print_avg_min_max (config, stats->sig_1_and_2);
  printf (", ");
  flp_analysis_print_avg_min_max (config, stats->n_mature);
  printf (", ");
  flp_analysis_print_avg_min_max (config, stats->n_memory);
  printf (":");

  flp_analysis_responses_print (stats, config);
  printf ("\n");

  return;
}

inline long double
flp_analysis_mean (const unsigned long long *data, const unsigned int len)
{
  unsigned int n;
  unsigned long long sum = 0ULL;

  for (n = 0; n < len; n++)
    {
      sum += data[n];
    }

  return (sum / (long double) len);
}

inline long double
flp_analysis_stddev (const unsigned long long *data, const unsigned int len,
                     const long double mean)
{
  register unsigned int n;
  unsigned long long sum = 0ULL;

  for (n = 0; n < len; n++)
    {
      /* Sum of all squared distances from mean. */
      sum += ((data[n] - mean) * (data[n] - mean));
    }

  return (sqrt ((long double) sum / ((long double) len - 1.0)));
}

inline void
flp_analysis_print_mean_stddev (const unsigned long long *val,
                                 const ais_config_t *config)
{
  long double mean = flp_analysis_mean (val, config->nodes);
  printf ("%.5Lf %.5Lf",
          mean,
          flp_analysis_stddev (val, config->nodes, mean));

  return;
}

inline int
flp_ull_cmp (const void *a, const void *b)
{
  const unsigned long long *ulla = (const unsigned long long *) a;
  const unsigned long long *ullb = (const unsigned long long *) b;

  return (*ulla > *ullb) - (*ulla < *ullb);
}

inline unsigned int
flp_is_odd (const unsigned int n)
{
  return ((n & 0x01) == 1) ? 1 : 0;
}

inline long double
flp_median_ull (const unsigned long long *v, const unsigned int n)
{
  long double median = 0.0;

  if (flp_is_odd (n))
    {
      unsigned int m = n / 2;
      median = v[m];
    }
  else
    {
      unsigned int m = (n / 2) - 1;
      median = ((long double) (v[m] + v[m + 1])) / 2.0;
    }

  return median;
}

/* Function to summarize an array of observations as a five-number
   summary. */
void
flp_analysis_print_5_numbers (const unsigned long long *val,
                               const ais_config_t *config)
{
  register unsigned int n;
  unsigned int q3_offset = 0;
  unsigned long long v[config->nodes];
  /* The five-number summary variables. */
  long double min = 0.0;
  long double q1 = 0.0;
  long double median = 0.0;
```

```
    long double q3 = 0.0;
    long double max = 0.0;

    for (n = 0; n < config->nodes; n++)
      {
        v[n] = val[n];
      }

    n = config->nodes;

    /* Arrange the observations from low to high, based on their
       values. */
    qsort (v, n, sizeof (*v), flp_ull_cmp);

    min = v[0];
    max = v[n - 1];

    median = flp_median_ull (&v[0], n);

    if (flp_is_odd (n))
      {
        q3_offset = (n / 2) + 1;;
      }
    else
      {
        q3_offset = (n / 2);
      }

    /* Number of elements in to the left of the median. */
    n = n / 2;

    q1 = flp_median_ull (&v[0], n);
    q3 = flp_median_ull (&v[q3_offset], n);

    printf ("%Lg %Lg %Lg %Lg %Lg", min, q1, median, q3, max);

    return;
  }

  void
  flp_analysis_5_numbers_print (const statistics_t *stats,
                                const ais_config_t *config)
  {
    flp_analysis_print_5_numbers(stats->premature_deaths, config);
    printf (", ");

    flp_analysis_print_5_numbers(stats->false_negatives, config);
    printf (" %qu, ", stats->b_false_negatives);

    flp_analysis_print_5_numbers(stats->true_negatives, config);
    printf (" %qu, ", stats->b_true_negatives);

    flp_analysis_print_5_numbers(stats->false_positives, config);
    printf (" %qu, ", stats->b_false_positives);

    flp_analysis_print_5_numbers(stats->true_positives, config);
    printf (" %qu, ", stats->b_true_positives);
  flp_analysis_print_5_numbers(stats->activations, config);
  printf (", ");
  flp_analysis_print_5_numbers(stats->sig_1_only, config);
  printf (", ");
  flp_analysis_print_5_numbers(stats->sig_1_and_2, config);
  printf (", ");
  flp_analysis_print_5_numbers(stats->n_mature, config);
  printf (", ");
  flp_analysis_print_5_numbers(stats->n_memory, config);
  printf (":");

  flp_analysis_responses_print (stats, config);
  printf ("\n");

  return;
}

void
flp_analysis_stddev_print (const statistics_t *stats,
                           const ais_config_t *config)
{
  /* Probably field 16. */
  flp_analysis_print_mean_stddev (stats->premature_deaths, config);
  printf (", ");

  /* Probably field 18. */
  flp_analysis_print_mean_stddev (stats->false_negatives, config);
  printf (" %qu, ", stats->b_false_negatives);

  /* Probably field 21. */
  flp_analysis_print_mean_stddev (stats->true_negatives, config);
  printf (" %qu, ", stats->b_true_negatives);

  /* Probably field 24. */
  flp_analysis_print_mean_stddev (stats->false_positives, config);
  printf (" %qu, ", stats->b_false_positives);

  /* Probably field 27. */
  flp_analysis_print_mean_stddev (stats->true_positives, config);
  printf (" %qu, ", stats->b_true_positives);

  /* Probably field 30. */
  flp_analysis_print_mean_stddev (stats->activations, config);
  printf (", ");
  /* Probably field 32. */
  flp_analysis_print_mean_stddev (stats->sig_1_only, config);
  printf (", ");
  /* Probably field 34. */
  flp_analysis_print_mean_stddev (stats->sig_1_and_2, config);
  printf (", ");
  /* Probably field 36. */
  flp_analysis_print_mean_stddev (stats->n_mature, config);
  printf (", ");
  /* Probably field 38. */
  flp_analysis_print_mean_stddev (stats->n_memory, config);
  printf (":");
  /* Probably field 40+. */
  flp_analysis_responses_print (stats, config);
  printf ("\n");

  return;
}

inline void
flp_statistics_destroy (statistics_t *stats)
{
  free (stats->false_alarms);
  free (stats->true_alarms);
  free (stats->n_memory);
  free (stats->n_mature);
  free (stats->sig_1_only);
  free (stats->sig_1_and_2);
  free (stats->activations);
  free (stats->false_negatives);
  free (stats->false_positives);
  free (stats->true_negatives);
  free (stats->true_positives);
  free (stats->premature_deaths);

  return;
}

inline void
flp_statistics_reset (statistics_t *stats, const ais_config_t *config)
{
  register unsigned int n, f;

  for (n = 0; n < config->nodes; n++)
    {
      stats->premature_deaths[n] = 0ULL;
      stats->true_positives[n] = 0ULL;
      stats->true_negatives[n] = 0ULL;
      stats->false_positives[n] = 0ULL;
      stats->false_negatives[n] = 0ULL;
      stats->activations[n] = 0ULL;
      stats->sig_1_and_2[n] = 0ULL;
      stats->sig_1_only[n] = 0ULL;

      stats->n_mature[n] = 0ULL;
      stats->n_memory[n] = 0ULL;
    }

  for (f = 0; f < config->file_names; f++)
    {
      stats->true_alarms[f] = 0;
      stats->false_alarms[f] = 0;
    }

  stats->b_true_positives = 0ULL;
  stats->b_true_negatives = 0ULL;
  stats->b_false_positives = 0ULL;
  stats->b_false_negatives = 0ULL;
  return;
}

inline void
flp_statistics_initialize (statistics_t *stats, const ais_config_t *config)
{
  stats->premature_deaths =
    (unsigned long long *) malloc (sizeof (*(stats->premature_deaths)) *
                                   config->nodes + 1);
  if (stats->premature_deaths == NULL)
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  stats->true_positives =
    (unsigned long long *) malloc (sizeof (*(stats->true_positives)) *
                                   config->nodes);
  if (stats->true_positives == NULL)
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  stats->true_negatives =
    (unsigned long long *) malloc (sizeof (*(stats->true_negatives)) *
                                   config->nodes);
  if (stats->true_negatives == NULL)
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  stats->false_positives =
    (unsigned long long *) malloc (sizeof (*(stats->false_positives)) *
                                   config->nodes);
  if (stats->false_positives == NULL)
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  stats->false_negatives =
    (unsigned long long *) malloc (sizeof (*(stats->false_negatives)) *
                                   config->nodes);
  if (stats->false_negatives == NULL)
    {
      perror (__FUNCTION__ ": virtual memory exhausted");
      exit (1);
    }

  stats->activations =
    (unsigned long long *) malloc (sizeof (*(stats->activations)) *
                                   config->nodes);
  if (stats->activations == NULL)
    {
        perror (__FUNCTION__ ": virtual memory exhausted");
        exit (1);
      }

    stats->sig_1_and_2 =
      (unsigned long long *) malloc (sizeof (*(stats->sig_1_and_2)) *
                                     config->nodes);
    if (stats->sig_1_and_2 == NULL)
      {
        perror (__FUNCTION__ ": virtual memory exhausted");
        exit (1);
      }

    stats->sig_1_only =
      (unsigned long long *) malloc (sizeof (*(stats->sig_1_only)) *
                                     config->nodes);
    if (stats->sig_1_only == NULL)
      {
        perror (__FUNCTION__ ": virtual memory exhausted");
        exit (1);
      }

    stats->n_mature =
      (unsigned long long *) malloc (sizeof (*(stats->n_mature)) *
                                     config->nodes);
    if (stats->n_mature == NULL)
      {
        perror (__FUNCTION__ ": virtual memory exhausted");
        exit (1);
      }

    stats->n_memory =
      (unsigned long long *) malloc (sizeof (*(stats->n_memory)) *
                                     config->nodes);
    if (stats->n_memory == NULL)
      {
        perror (__FUNCTION__ ": virtual memory exhausted");
        exit (1);
      }

    stats->true_alarms =
      (unsigned int *) malloc (sizeof (*(stats->true_alarms))
                               * config->file_names);
    if (stats->true_alarms == NULL)
      {
        perror (__FUNCTION__ ": virtual memory exhausted");
        exit (1);
      }

    stats->false_alarms =
      (unsigned int *) malloc (sizeof (*(stats->false_alarms))
                               * config->file_names);
    if (stats->false_alarms == NULL)
      {
        perror (__FUNCTION__ ": virtual memory exhausted");
        exit (1);
      }
```

```
  flp_statistics_reset (stats, config);

  return;
}

char *
strconcat (const char *str, ...)
{
  va_list ap;
  size_t allocated = 256;
  char *result = (char *) malloc (allocated);
  char *wp;

  if (result != NULL)
    {
      char *newp;
      const char *s;

      va_start (ap, str);

      wp = result;
      for (s = str; s != NULL; s = va_arg (ap, const char *))
        {
          size_t len = strlen (s);

          /* Resize the allocated memory if necessary. */
          if (wp + len + 1 > result + allocated)
            {
              allocated = (allocated + len) * 2;
              newp = (char *) realloc (result, allocated);
              if (newp == NULL)
                {
                  free (result);
                  return NULL;
                }
              wp = newp + (wp − result);
              result = newp;
            }
          wp = MEMPCPY (wp, s, len);
        }

      /* Terminate the result string. */
      *wp++ = '\0';

      /* Resize memory to the optimal size. */
      newp = realloc (result, wp − result);
      if (newp != NULL)
        result = newp;

      va_end (ap);
    }

  return result;
}

char
```

```
flp_data_is_self (const char *data_dir, const ais_config_t *config,
                  int *dfile, FILE **data)
{
  char dbuf[FGETS_BUF_SIZE] = {0, };
  char self_nonself = '\0';

  if (data_dir != NULL)
    {
      if ((*dfile) == -1)
        {
          /* Open the first file. */
          char *tmp = strconcat (data_dir, config->file_name_list[++(*dfile)]);
          //printf ("Opening %s\n", tmp);
          *data = flp_file_open (tmp);
          free (tmp);
        }

      if (NULL == fgets (dbuf, FGETS_BUF_SIZE, *data))
        {
          /* End of file. */
          flp_file_close (*data);

          if (++(*dfile) < config->file_names)
            {
              /* Open next file. */
              char *tmp = strconcat (data_dir,
                                     config->file_name_list[*(dfile)]);
              //printf ("Opening %s\n", tmp);
              *data = flp_file_open (tmp);
              free (tmp);

              if (NULL == fgets (dbuf, FGETS_BUF_SIZE, *data))
                {
                  perror (__FUNCTION__
                          "Where the shit hits the fan!\n");
                  exit (1);
                }
            }
        }
      //printf ("This went OK!\n");

      self_nonself = dbuf[0];
    }

  return self_nonself;
}


statistics_t *
flp_parse (FILE *stream, const ais_config_t *config,
           const unsigned long long offset, const char *data_dir)
{
  statistics_t *stats = (statistics_t *) malloc (sizeof(*stats));
  unsigned int done = FALSE;
  int dfile = -1;
  const char *format = " %qu %qu %qu %qu %qu %qu %qu %qu %qu %qu %qu";
  unsigned long long line = 0ULL;
```

```
FILE *data = NULL;
char *mem = (char *) malloc (sizeof (*mem) * FGETS_BUF_SIZE);
if (mem == NULL)
  {
    perror (__FUNCTION__ ": virtual memory exhausted");
    exit (1);
  }

flp_statistics_initialize (stats, config);

while (! done)
  {
    char buf[FGETS_BUF_SIZE] = {0,};

    if (NULL == fgets (buf, FGETS_BUF_SIZE, stream))
      {
        done = TRUE;
      }
    else
      {
        /* Check that it is neither a comment line nor a data line
           we're gonna use; i.e., with LINE < OFFSET. */
        if (buf[0] != '#')
          {
            char self_nonself = '\0';
            //printf (__FUNCTION__ ": Calling...");
            self_nonself = flp_data_is_self (data_dir, config,
                                             &dfile, &data);
            //printf (__FUNCTION__ ": done\n");
            //printf ("%c\n", self_nonself);

            if (++line > offset)
              {
                volatile unsigned long long y_activations = 0ULL;
                volatile unsigned long long y_true_positives = 0ULL;
                volatile unsigned long long y_true_negatives = 0ULL;
                volatile unsigned long long y_false_positives = 0ULL;
                volatile unsigned long long y_false_negatives = 0ULL;
                char *str = strcpy (mem, buf);
                char *tmp = strsep (&str, ":");
                volatile unsigned int file = atoi (tmp);
                register unsigned int n;

                for (n = 0; n < config->nodes; n++)
                  {
                    unsigned long long x_premature_deaths = 0ULL;
                    unsigned long long x_true_positives = 0ULL;
                    unsigned long long x_true_negatives = 0ULL;
                    unsigned long long x_false_positives = 0ULL;
                    unsigned long long x_false_negatives = 0ULL;
                    unsigned long long x_activations = 0ULL;
                    unsigned long long x_sig_1_and_2 = 0ULL;
                    unsigned long long x_sig_1_only = 0ULL;
                    unsigned long long x_n_mature = 0ULL;
                    unsigned long long x_n_memory = 0ULL;

                    tmp = strsep (&str, ":");
            sscanf (tmp, format,
                    &x_premature_deaths,
                    &x_false_negatives,
                    &x_true_negatives,
                    &x_false_positives,
                    &x_true_positives,
                    &x_activations,
                    &x_sig_1_only,
                    &x_sig_1_and_2,
                    &x_n_mature,
                    &x_n_memory);

            stats->premature_deaths[n] += x_premature_deaths;

            stats->false_negatives[n] += x_false_negatives;
            stats->true_negatives[n] += x_true_negatives;
            stats->false_positives[n] += x_false_positives;
            stats->true_positives[n] += x_true_positives;

            stats->activations[n] += x_activations;

            stats->sig_1_only[n] += x_sig_1_only;
            stats->sig_1_and_2[n] += x_sig_1_and_2;

            stats->n_mature[n] += x_n_mature;
            stats->n_memory[n] += x_n_memory;

            y_false_negatives += x_false_negatives;
            y_true_negatives += x_true_negatives;
            y_false_positives += x_false_positives;
            y_true_positives += x_true_positives;

            y_activations += x_activations;
          }

        /* Boolean count of true and false positives and
           negatives.   I.e., if any of the nodes classified the
           agent as any of these classes, increase the B_counter
           by _one_. */
        stats->b_false_negatives += y_false_negatives ? 1 : 0;
        stats->b_true_negatives += y_true_negatives ? 1 : 0;
        stats->b_false_positives += y_false_positives ? 1 : 0;
        stats->b_true_positives += y_true_positives ? 1 : 0;

        /* Similarly possibly increase the true and false alarms
           counter.   This is recording if an actual activation
           was reached. */

        if (y_activations)
          {
            if (self_nonself == 'S')
              {
                stats->false_alarms[file]++;
              }
            else if (self_nonself == 'N')
              {
                stats->true_alarms[file]++;
```

```
                        }
                      else
                        {
                          fprintf (stderr, "Gokkiguoppi!\n");
                        }
                    }
                }
            }
          else
            {
              /* Comment line in the log file.      Thus, no reading
                 should be done from the data file. */
            }
        }
    }

  free (mem);

  return stats;
}
```

Appendix B

GNU General Public License

B.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

B.2 Terms and Conditions for Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel,

ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

B.3 How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program’s name and a brief idea of what it does.>
Copyright (C) <year>  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
‘Gnomovision’ (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

References

Balthrop, Justin, Stephanie Forrest, and Matthew R. Glickman. 2002. "Revisiting LISYS: Parameters and Normal Behavior." *Proceedings of the 2002 Congress on Evolutionary Computation*. In press.

Burgess, Mark. 1998, December 6–11. "Computer Immunology." *Proceedings of the Twelfth Systems Administration Conference (LISA 98)*. Boston, Massachusetts, USA: USENIX, 283–298.

CERT Coordination Center. 2002, April. "CERT/CC Statistics 1988-2002." Technical Report, Carnegie Mellon University. Available online at `<URL:http://www.cert.org/stats/cert_stats.html>`.

Cho, Sung-Bae. 2000. "Artificial Life Technology for Adaptive Information Processing." Chapter 2 in *Future Directions for Intelligent Systems and Information Sciences: The Future of Speech and Image Technologies, Brain Computers, WWW, and Bioinformatics*, edited by Nikola Kasabov, Volume 45 of *Studies in Fuzziness and Soft Computing*, 13–33. Heidelberg, Germany: Physica-Verlag. ISBN 3-7908-1276-5.

Dasgupta, Dipankar. 1999, October. "Immunity-Based Intrusion Detection System: A General Framework." *Proceedings of the 22nd National Information Systems Security Conference (NISSC)*. National Institute of Standards and Technology and National Computer Security Center, Hyatt Regency — Crystal City, Virginia, United States.

Dasgupta, Dipankar, Yuehua Cao, and Congjun Yang. 1999, July 13–17. "An Immunogenetic Approach to Spectra Recognition." Edited by Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, *Proceedings of the Genetic and Evolutionary Computation (GECCO) Conference*, Volume 1. Orlando, Florida, United States: Morgan Kaufmann, 149–155. ISBN 1-55860-611-4.

Dasgupta, Dipankar, and Stephanie Forrest. 1996, June 19–21. "Novelty Detection in Time Series Data using Ideas from Immunology." *Proceedings of the 5th International Conference on Intelligent Systems*. Reno, Nevada, United States.

Feller, William. 1968a. "Binomial and the Poisson Distributions." Chap. VI in *An Introduction to Probability Theory and Its Applications*, Volume 1 of *Wiley Series in Probability and Mathematical Statistics*, 3d ed., 146–173. New York, United States: John Wiley & Sons. ISBN 0-471-25708-7.

———. 1968b. "Recurrent Events. Renewal Theory." Chap. XIII in *An Introduction to Probability Theory and Its Applications*, Volume 1 of *Wiley Series in Probability and Mathematical Statistics*, 3d ed., 303–341. New York, United States: John Wiley & Sons. ISBN 0-471-25708-7.

Forrest, Stephanie, and Steven Andrew Hofmeyr. 2001, July. "Immunology as Information Processing." Chapter in *Design Principles for the Immune System and Other Distributed Autonomous Systems*, edited by Lee A. Segel and Irun R. Cohen, Santa Fe Institute Studies on the Sciences of Complexity. New York, United States: Oxford University Press. ISBN 0-19-513699-3.

Forrest, Stephanie, Steven Andrew Hofmeyr, and Anil Somayaji. 1997. "Computer Immunology." *Communications of the ACM* 40 (10): 88–96 (October).

Forrest, Stephanie, Steven Andrew Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. 1996, May. "A Sense of Self for Unix Processes." *Proceedings of the 1996 IEEE Symposium on Security and Privacy*. Los Alamitos, California, United States: IEEE Computer Society Press, 120–128.

Forrest, Stephanie, Alan S. Perelson, Lawrence Allen, and Rajesh Cherukuri. 1994, May. "Self-Nonself Discrimination in a Computer." *1994 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society Technical Committee in Security and privacy and The International Association for Cryptologic Research (IACR), Oakland, California, United States: IEEE Computer Society Press, 202–212. ISBN 0-8186-5675-1.

Halsall, Fred. 1996. *Data Communications, Computer Networks and Open Systems*. 4th ed. Electronic Systems Engineering Series. United States: Addison-Wesley Publishing Company. ISBN 0-201-42293-X.

Hightower, Ron, Stephanie Forrest, and Alan S. Perelson. 1996. "The Baldwin Effect in the Immune System: Learning by Somatic Hypermutation." Edited by Richard K. Belew and Melanie Mitchell, *Adaptive Individuals in Evolving Populations: Models and Algorithms*, Santa Fe Institute (SFI) Studies in the Sciences of Complexity. Addison-Wesley Publishing Company, Reading Massachusetts, 159–167.

Hofmeyr, Steven Andrew. 1999, May. "An Immunological Model of Distributed Detection and Its Application to Computer Security." Ph. D. dissertation, University of New Mexico, Albuquerque, NM 87131-1386, United States.

Hofmeyr, Steven Andrew, and Stephanie Forrest. 2000. "Architecture for an Artificial Immune System." *Evolutionary Computation* 8 (4): 443–473.

Householder, Allen, Kevin Houle, and Chad Dougherty. 2002. "Computer Attack Trends Challenge Internet Security." *Security & Privacy* (supplement to *Computer Magazine*), April, 5–7.

Kemmerer, Richard A., and Giovanni Vigna. 2002. "Intrusion Detection: A Brief History and Overview." *Security & Privacy* (supplement to *Computer Magazine*), April, 27–29.

Kim, Jungwon, and Peter J. Bentley. 2001, July. "An Evaluation of Negative Selection in an Artificial Immune System for Network Intrusion Detection." *Proceedings of the Genetic and Evolutionary Computation Conference 2001 (GECCO-2001)*. San Francisco, United States, 1330–1337.

Knuth, Donald Ervin. 1997a, September. "Mathematical Preliminaries." Section. 1.2 in *Fundamental Algorithms*, Volume 1 of *The Art of Computer Programming*, 3d ed., 10–123. Reading Massachusetts: Addison-Wesley Publishing Company. ISBN 0-201-89683-4.

———. 1997b, September. "Dynamic Storage Allocation." Section 2.5 in *Fundamental Algorithms*, Volume 1 of *The Art of Computer Programming*, 3d ed., 435–456. Reading Massachusetts: Addison-Wesley Publishing Company. ISBN 0-201-89683-4.

Matsumoto, Makoto, and Takuji Nishimura. 1998. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator." *ACM Transactions on Modeling and Computer Simulation* 8 (1): 3–30 (January). ISSN 1049-3301.

McHugh, John. 2000. "Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by Lincoln Laboratory." *ACM Transactions on Information and System Security (TISSEC)* 3 (4): 262–294 (November). ISSN 1094-9224.

McHugh, John, Alan Christie, and Julia Allen. 2000. "Defending Yourself: The Role of Intrusion Detection Systems." *IEEE Software* 17 (5): 42–51 (September/October). ISSN 0740-7459.

Michalewicz, Zbigniew, and David B. Fogel. 2000. *How to Solve It: Modern Heuristics*. Berlin Heidelberg, Germany: Springer-Verlag. Corrected Second Printing 2000.

Muselli, Marco. 1996. "Simple expressions for success run distributions in Bernoulli trials." *Statistics & Probability Letters* 31 (2): 121–128 (16 December).

Navarro, Gonzalo. 2001. "A Guided Tour to Approximate String Matching." *ACM Computing Surveys (CSUR)* 33 (1): 31–88 (March). ISSN 0360-0300.

Northcutt, Stephen, Mark Cooper, Matt Fearnow, and Karen Frederick. 2001, January. *Intrusion Signatures and Analysis*. 1st ed. 201 West 103rd Street, Indianapolis, Indiana 46290, United States: New Riders Publishing. ISBN 0-7357-1063-5.

Northcutt, Stephen, Judy Novak, and Donald McLachlan. 2000, September. *Network Intrusion Detection: An Analyst's Handbook*. 2d ed. 201 West 103rd Street, Indianapolis, Indiana 46290, United States: New Riders Publishing. ISBN 0-7357-1008-2.

Percus, Jerome K., Ora E. Percus, and Alan S. Perelson. 1993, March. "Predicting the size of the T-cell receptor and antibody combining region from consideration of efficient self-nonself discrimination." *Proceedings of the National Academy of Sciences of the United States of America*, Volume 90. 1691–1695.

Postel, Jonathan B. 1981a, September. "Internet Protocol." RFC 791, The Internet Engineering Task Force (IETF).

———. 1981b, September. "Transmission Control Protocol." RFC 793, The Internet Engineering Task Force (IETF).

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. 1992. Random Numbers. Chap. 7 in *Numerical Recipes in C: The Art of Scientific Computing*, 2d ed., 274–328. 40 West 20th Street, New York, NY 10011-4211, United States: Cambridge University Press. ISBN 0-521-43108-5.

Reynolds, Joyce K. 2002, January. "Assigned Numbers: RFC 1700 is Replaced by an On-line Database." RFC 3232, The Internet Engineering Task Force (IETF).

Roitt, Ivan M., and Peter J. Delves. 2001. *Roitt's Essential Immunology*. 10th ed. Edited by Nick Morgan, Meg Barton, and Fiona Goodgame. Commerce Place, 350 Main Street, Malden, MA 02148-5018, USA: Blackwell Science. ISBN 0-632-05902-8.

Somayaji, Anil, and Stephanie Forrest. 2000, August 14–17. "Automated Response Using System-Call Delays." *Proceedings of the 9th USENIX Security Symposium*, USENIX Security Symposium. USENIX, Denver, Colorado, USA: USENIX.

Somayaji, Anil, Steven Andrew Hofmeyr, and Stephanie Forrest. 1998. "Principles of a Computer Immune System." *Proceedings of the workshop on New security paradigms workshop 1997*, New Security Paradigms Workshop. ACM Special Interest Group on Security, Audit, and Control: ACM Press, New York, NY, United States, 75–82. ISBN 0-89791-986-6.

Staniford, Stuart, Vern Paxson, and Nicholas Weaver. 2002, May. "How to 0wn the Internet in Your Spare Time." To Appear in the Proceedings of the 11th USENIX Security Symposium (Security '02).

Stillerman, Matthew, Carla Marceau, and Maureen Stillman. 1999. "Intrusion Detection for Distributed Applications." *Communications of the ACM* 42 (7): 62–69 (July).

Timmis, Jon, and Mark Neal. 2001. "A resource limited artificial immune system for data analysis." *Knowledge-Based Systems* 14 (3–4): 121–130 (June).

Timmis, Jon, Mark Neal, and John Hunt. 2000. "An artificial immune system for data analysis." *BioSystems* 55 (1–3): 143–150 (February).

Venter, J. Craig, et al. 2001. "The Sequence of the Human Genome." *Science* 291 (5507): 1304–1351 (February 16).

Wilf, Herbert S. 1993, November. *generatingfunctionology*. 2d ed. Boston, United States: Academic Press, Inc. ISBN 0-12-751956-4, Available online from `<URL:http://www.math.upenn.edu/~wilf/DownldGF.html>`.

Williams, Paul D., Kevin P. Anchor, John L. Bebo, Gregg H. Gunsch, and Gary D. Lamont. 2001, October. "CDIS: Towards a Computer Immune System for Detecting Network Intrusions." Edited by Wenke Lee, Ludovic Mé, and Andreas Wespi, *Recent Advances in Intrusion Detection: 4th International Symposium; RAID 2001 Davis, CA, USA, October 10–12, 2001, Proceedings*, Volume 2212 of *Lecture Notes in Computer Science*. Davis, California, United States: Springer-Verlag, Berlin Heidelberg, Germany, 117–133. ISBN 3-540-42702-3.

Function Index

Index